Contents

1 Overview **3**
1.1 Features - Resin and Resin Professional 3

2 Installation **11**
2.1 Resin Installation Quick Start 11
2.2 Resin Installation 16
2.3 Resin Web Server 16
2.4 Resin with Apache 22
2.5 Resin with IIS 34
2.6 How the Plugins Dispatch to Resin 44

3 Command-Line **47**
3.1 Command-Line Configuration 47

4 Admin Guide **51**
4.1 User Guide: Administration 51

5 Watchdog **63**
5.1 Resin Watchdog 63

6 Virtual Hosts **73**
6.1 Virtual Hosting 73

7 Clustering **89**
7.1 Resin Clustering 89

8 Web Applications **109**
8.1 An Overview of Web Applications 109

9 Logging **137**
9.1 Log 137

10 Administration **163**
10.1 Resin Administration 163

11 Deployment **177**
11.1 Packaging/Deployment 177

12 Proxy Caching **181**
12.1 Server Caching 181

13 Quercus **193**
13.1 Quercus: PHP in Java 193

14 Security **217**
14.1 Resin Security 217

15 Inversion of Control **271**
15.1 Resin IoC 271
15.2 Scheduled Task 308

16 Amber **327**
16.1 Amber 327

17 Embedding Resin **355**
17.1 Embedding Resin 355

18 Filters **367**
18.1 Filters 367

19 BAM **379**
19.1 BAM 379

20 Comet **405**
20.1 Comet/Server-Push Servlet 405

21 Remoting **411**
21.1 Resin Remoting 411
21.2 Hessian 417

22 Messaging **423**
22.1 Resin Messaging 423

23 JSF - Java Server Faces **435**
23.1 JSF - Java Server Faces 435

24 Configuration Tags **445**
24.1 cluster: Cluster tag configuration 445
24.1.1 <access-log> 445
24.1.2 <cache> 445
24.1.3 <cluster> 447
24.1.4 <cluster-default> 450
24.1.5 <connection-error-page> 450

24.1.6 <development-mode-error-page> . . . 451
24.1.7 <ear-default> . . . 451
24.1.8 <error-page> . . . 451
24.1.9 <host> . . . 451
24.1.10 <host-default> . . . 452
24.1.11 <host-deploy> . . . 452
24.1.12 <ignore-client-disconnect> . . . 453
24.1.13 <invocation-cache-size> . . . 453
24.1.14 <invocation-cache-max-url-length> . . . 453
24.1.15 <persistent-store> . . . 454
24.1.16 <ping> . . . 457
24.1.17 Resource Tags . . . 459
24.1.18 <rewrite-dispatch> . . . 459
24.1.19 <root-directory> . . . 460
24.1.20 <server> . . . 460
24.1.21 <server-default> . . . 465
24.1.22 <server-header> . . . 465
24.1.23 <session-cookie> . . . 466
24.1.24 <session-sticky-disable> . . . 466
24.1.25 <session-url-prefix> . . . 466
24.1.26 <ssl-session-cookie> . . . 467
24.1.27 <url-character-encoding> . . . 467
24.1.28 <web-app-default> . . . 467
24.2 database: Database tag configuration . . . 468
24.2.1 <connection-wait-time> . . . 468
24.2.2 <close-dangling-connections> . . . 468
24.2.3 <driver> . . . 468
24.2.4 <database> . . . 469
24.2.5 <max-active-time> . . . 470
24.2.6 <max-close-statements> . . . 470
24.2.7 <max-connections> . . . 470
24.2.8 <max-create-connections> . . . 471
24.2.9 <max-idle-time> . . . 471
24.2.10 <max-pool-time> . . . 471
24.2.11 <password> . . . 471
24.2.12 <ping> . . . 471
24.2.13 <ping-table> . . . 472
24.2.14 <ping-query> . . . 472
24.2.15 <ping-interval> . . . 472
24.2.16 <prepared-statement-cache-size> . . . 472
24.2.17 <save-allocation-stack-trace> . . . 472
24.2.18 <spy> . . . 473
24.2.19 <transaction-timeout> . . . 473
24.3 <host>: Virtual Host configuration . . . 473
24.3.1 <access-log> . . . 473
24.3.2 <ear-deploy> . . . 476

24.3.3 <error-page> . . . 477
24.3.4 <host> . . . 477
24.3.5 <host-alias> . . . 479
24.3.6 <host-alias-regexp> . . . 479
24.3.7 <host-default> . . . 480
24.3.8 <host-deploy> . . . 480
24.3.9 <host-name> . . . 481
24.3.10 <redeploy-mode> . . . 481
24.3.11 Resources . . . 482
24.3.12 <rewrite-dispatch> . . . 482
24.3.13 <root-directory> . . . 483
24.3.14 <secure-host-name> . . . 483
24.3.15 <startup-mode> . . . 483
24.3.16 <web-app> . . . 484
24.3.17 <web-app-default> . . . 486
24.3.18 <web-app-deploy> . . . 486
24.4 port: Port tag configuration . . . 489
24.4.1 <accept-listen-backlog> . . . 489
24.4.2 <accept-thread-max> . . . 489
24.4.3 <accept-thread-min> . . . 489
24.4.4 <address> . . . 489
24.4.5 <ca-certificate-file> (OpenSSL) . . . 490
24.4.6 <ca-certificate-path> (OpenSSL) . . . 490
24.4.7 <ca-revocation-file> (OpenSSL) . . . 490
24.4.8 <ca-revocation-path> (OpenSSL) . . . 490
24.4.9 <certificate-file> (OpenSSL) . . . 490
24.4.10 <certificate-chain-file> (OpenSSL) . . . 490
24.4.11 <certificate-key-file> (OpenSSL) . . . 490
24.4.12 <cipher-suite> (OpenSSL) . . . 490
24.4.13 <cluster-port> . . . 491
24.4.14 <connection-max> . . . 491
24.4.15 <http> . . . 491
24.4.16 <jsse-ssl> . . . 492
24.4.17 <keepalive-max> . . . 493
24.4.18 <openssl> . . . 493
24.4.19 <password> (OpenSSL) . . . 494
24.4.20 <port> . . . 494
24.4.21 <protocol> . . . 494
24.4.22 <protocol> (OpenSSL) . . . 495
24.4.23 <session-cache> (OpenSSL) . . . 495
24.4.24 <session-cache-timeout> (OpenSSL) . . . 495
24.4.25 <socket-timeout> . . . 495
24.4.26 <tcp-no-delay> . . . 495
24.4.27 <unclean-shutdown> (OpenSSL) . . . 495
24.4.28 <verify-client> (OpenSSL) . . . 496
24.4.29 <verify-depth> (OpenSSL) . . . 496

24.5 Resources: class loaders, environment and IoC 496
24.5.1 <authenticator> 496
24.5.2 <bam-service> 498
24.5.3 <bean> 499
24.5.4 <case-insensitive> 500
24.5.5 <character-encoding> 501
24.5.6 <class-loader> 502
24.5.7 <compiling-loader> 503
24.5.8 <component> 504
24.5.9 <connection-factory> 505
24.5.10 <database> 507
24.5.11 <database-default> 512
24.5.12 <dependency> 513
24.5.13 <dependency-check-interval> 514
24.5.14 <ejb-message-bean> 514
24.5.15 <ejb-server> 516
24.5.16 <ejb-stateful-bean> 517
24.5.17 <ejb-stateless-bean> 518
24.5.18 <env-entry> 519
24.5.19 <fileset> 521
24.5.20 <javac> 523
24.5.21 <jms-connection-factory> 525
24.5.22 <jms-queue> 525
24.5.23 <jms-topic> 526
24.5.24 <jndi-link> 527
24.5.25 <jpa-persistence> 529
24.5.26 <library-loader> 530
24.5.27 <log> 531
24.5.28 <logger> 533
24.5.29 <mail> 533
24.5.30 <reference> 535
24.5.31 <remote-client> 536
24.5.32 <resin:choose> 537
24.5.33 <resin:if> 541
24.5.34 <resin:import> 541
24.5.35 <resin:message> 544
24.5.36 <resin:set> 544
24.5.37 <resource> 545
24.5.38 <resource-adapter> 546
24.5.39 <resource-deploy> 547
24.5.40 <resource-ref> 548
24.5.41 <scheduled-task> 549
24.5.42 <servlet-hack> 551
24.5.43 <simple-loader> 552
24.5.44 <stderr-log> 553
24.5.45 <stdout-log> 554

24.5.46 <system-property> . . . 555
24.5.47 <temp-dir> . . . 556
24.5.48 <tree-loader> . . . 557
24.5.49 <work-dir> . . . 558
24.6 URL rewrite tags . . . 559
24.6.1 <and> . . . 559
24.6.2 auth-type . . . 559
24.6.3 cookie . . . 559
24.6.4 disable-at . . . 560
24.6.5 <dispatch> . . . 560
24.6.6 dispatch-type . . . 561
24.6.7 enable-at . . . 561
24.6.8 enabled . . . 561
24.6.9 <forbidden> . . . 561
24.6.10 <forward> . . . 562
24.6.11 header . . . 563
24.6.12 <gone> . . . 563
24.6.13 <import> . . . 564
24.6.14 <load-balance> . . . 566
24.6.15 locale . . . 568
24.6.16 local-port . . . 568
24.6.17 method . . . 569
24.6.18 <match> . . . 569
24.6.19 <moved-permanently> . . . 570
24.6.20 name . . . 571
24.6.21 <not> . . . 571
24.6.22 <or> . . . 571
24.6.23 <not-found> . . . 571
24.6.24 query-param . . . 572
24.6.25 <redirect> . . . 572
24.6.26 <regexp> . . . 574
24.6.27 remote-addr . . . 574
24.6.28 remote-user . . . 574
24.6.29 <rewrite> . . . 575
24.6.30 <rewrite-real-path> . . . 575
24.6.31 <rewrite-dispatch> . . . 576
24.6.32 secure . . . 577
24.6.33 server-name . . . 578
24.6.34 server-port . . . 578
24.6.35 <set> . . . 579
24.6.36 <unless> . . . 580
24.6.37 user-in-role . . . 580
24.6.38 <when> . . . 581
24.7 server: Server tag configuration . . . 584
24.7.1 <accept-listen-backlog> . . . 585
24.7.2 <accept-thread-max> . . . 585

24.7.3 <accept-thread-min> . . . 585
24.7.4 <address> . . . 585
24.7.5 <cluster-port> . . . 586
24.7.6 <group-name> . . . 587
24.7.7 <http> . . . 587
24.7.8 <jvm-arg> . . . 588
24.7.9 <jvm-classpath> . . . 589
24.7.10 <keepalive-max> . . . 589
24.7.11 <keepalive-select-enable> . . . 590
24.7.12 <keepalive-select-thread-timeout> . . . 590
24.7.13 <keepalive-timeout> . . . 590
24.7.14 <load-balance-connect-timeout> . . . 591
24.7.15 <load-balance-recover-time> . . . 591
24.7.16 <load-balance-idle-time> . . . 592
24.7.17 <load-balance-warmup-time> . . . 592
24.7.18 <load-balance-weight> . . . 593
24.7.19 <memory-free-min> . . . 593
24.7.20 <port> . . . 594
24.7.21 <protocol> . . . 595
24.7.22 <server> . . . 595
24.7.23 <server-default> . . . 597
24.7.24 <shutdown-wait-max> . . . 598
24.7.25 <socket-timeout> . . . 598
24.7.26 <thread-idle-max> . . . 599
24.7.27 <thread-idle-min> . . . 599
24.7.28 <thread-max> . . . 600
24.7.29 <user-name> . . . 601
24.7.30 <watchdog-arg> . . . 601
24.7.31 <watchdog-port> . . . 602
24.8 Session tags . . . 602
24.8.1 <cookie-domain> . . . 602
24.8.2 <cookie-length> . . . 603
24.8.3 <cookie-max-age> . . . 603
24.8.4 <cookie-version> . . . 603
24.8.5 <ignore-serialization-errors> . . . 603
24.8.6 <session-config> . . . 604
24.8.7 <session-max> . . . 606
24.8.8 <save-mode> . . . 606
24.8.9 <session-timeout> . . . 606
24.9 EL Variables and Functions . . . 606
24.9.1 Environment variables . . . 607
24.9.2 fmt.sprintf() . . . 608
24.9.3 fmt.timestamp() . . . 610
24.9.4 host . . . 611
24.9.5 java . . . 611
24.9.6 jndi() . . . 611

24.9.7 resin . . . 612
24.9.8 server . . . 612
24.9.9 System properties . . . 613
24.9.10 webApp . . . 613
24.10 Web Application: tags . . . 613
24.10.1 <access-log> . . . 614
24.10.2 <active-wait-time> . . . 614
24.10.3 <allow-servlet-el> . . . 614
24.10.4 <archive-path> . . . 614
24.10.5 <auth-constraint> . . . 615
24.10.6 <cache-mapping> . . . 616
24.10.7 <constraint> . . . 617
24.10.8 <context-param> . . . 618
24.10.9 <cookie-http-only> . . . 619
24.10.10 <ear-deploy> . . . 619
24.10.11 <error-page> . . . 621
24.10.12 <filter> . . . 622
24.10.13 <filter-mapping> . . . 624
24.10.14 <form-login-config> . . . 626
24.10.15 <idle-time> . . . 628
24.10.16 <jsf> . . . 628
24.10.17 <jsp> . . . 630
24.10.18 <jsp-config> . . . 632
24.10.19 <lazy-servlet-validate> . . . 633
24.10.20 <listener> . . . 633
24.10.21 <login-config> . . . 634
24.10.22 <mime-mapping> . . . 634
24.10.23 <multipart-form> . . . 635
24.10.24 <path-mapping> . . . 636
24.10.25 <protocol> . . . 637
24.10.26 <redeploy-check-interval> . . . 638
24.10.27 <redeploy-mode> . . . 638
24.10.28 Resource Tags . . . 638
24.10.29 <rewrite-dispatch> . . . 639
24.10.30 <rewrite-real-path> . . . 640
24.10.31 <root-directory> . . . 641
24.10.32 <secure> . . . 641
24.10.33 <security-constraint> . . . 641
24.10.34 <servlet> . . . 642
24.10.35 <servlet-mapping> . . . 644
24.10.36 <servlet-regexp> . . . 646
24.10.37 <session-config> . . . 648
24.10.38 <shutdown-wait-max> . . . 650
24.10.39 <statistics-enable> . . . 651
24.10.40 <strict-mapping> . . . 651
24.10.41 <user-data-constraint> . . . 652

24.10.42<web-app> 653
24.10.43<web-app-default> 655
24.10.44<web-app-deploy> 656
24.10.45<web-resource-collection> 658
24.10.46<welcome-file-list> 659

Chapter 1

Overview

1.1 Features - Resin and Resin Professional

Resin Professional

Resin is provided in two versions, Resin Professional and Resin Open Source. Resin Professional adds features and enhancements commonly needed in a professional production environment. Resin Open Source is suitable for hobbyists, developers, and low traffic websites that do not need the performance and reliability enhancements of Resin Professional.

Reliability features

Resin Professional provides a number of reliability features, including automatic server restart, detection and restart of locked or stalled servers, and monitoring of JVM memory usage for applications with memory leaks.

- Resin configuration: <ping>
- Resin configuration: min-free-memory

Clustering

Clustering provides the ability for multiple servers to appear as one server to clients. Clustering provides enhanced reliability and allows sites to scale up as server demand increases,

Clustering is supported with the standalone web server, Apache, and IIS.

- Resin documentation: Reliability and Load Balancing

Persistent and Distributed sessions

Persistent sessions guarantee that a server can restore the contents of the HttpSession object when it is restarted. Distributed sessions provide the ability for multiple servers in a cluster to share the values stored in the HttpSession.

- Resin documentation: Persistent and Distributed Sessions

Performance enhancing native code

Resin Professional includes a native code libary on both Windows and Unix platforms. Native code is used to provide significant performance benefits in areas like socket connections, keepalive connections, and file system access.

OpenSSL

Resin Professional uses native code to link to the OpenSSL libraries, a much better and more efficient SSL solution than the Java facilities provided by JSSE.

- Resin documentation: OpenSSL

HTTP proxy caching

Resin Professional provides a memory and disk based caching system for increased performance benefits. Server caching can speed dynamic pages to near-static speeds. Small but frequently accessed resources such as images and css files are cached in memory and served directly to the client, avoiding even a read from the disk.

Many pages require expensive operations like database queries but change infrequently. Resin can cache the results and serve them like static pages. Resin's caching will work for any servlet, including JSP and XTP pages.

- Resin documentation: Caching
- Resin configuration: <cache>

Gzip filter

Bandwidth costs are significant for many websites. The GzipFilter enables automatic compression of responses for browsers that support it. Use of the GzipFilter reduces bandwidth usage and may provide significant cost savings.

- Resin documentation: GzipFilter

Web Server

Standalone

- Resin documentation: Standalone web server Unix and Windows

Hypertext Transfer Protocol (HTTP)

Secure Socket Layer (SSL) 3.0

- Resin documentation

Apache 2.0 and 2.2 integration

- Resin documentation: Resin with Apache

IIS 5 and IIS 6 integration

- Resin documentation: Resin with IIS

CGI

- Resin documentation

Resin-IoC and Dependency Injection

- See Resin-Ioc documentation.

Resin uses its own IoC/Dependency Injection engine for all of its JavaEE configuration. With the new WebBeans (JSR-299) specification draft, Resin provides the same capabilities to application code. Components and singleton beans can be configured in the resin-web.xml for full XML configurability, or scanned for the class annotations for lightweight injection to match the style of the application.

Because Resin-IoC is fully integrated with Resin's EJB 3.0 support, all application components and beans can use EJB aspects such as @TransactionAttribute, @Stateless, @InterceptorClass, and @AroundInvoke.

The WebBeans IoC capabilities provide finely-controlled interception with @InterceptorType binding, and type-safe dependency-injection with @BindingType annotations. Testing with mock components are automatically supported with custom @ComponentType annotations.

Event handling is fully integrated with the WebBeans API using a flexible @Observes parameter attribute, enabling any bean or component to listen to any matching typed event thrown through the WebBeans Container interface.

Resin-managed classes are automatically enlisted in the IoC engine. So Servlets, Filters, application listener classes, remote objects, and EJBs can all use Resin-IoC capabilities without any additional configuration.

Resin-IoC provides a straightforward driver for integrating Resin-IoC capabilities with popular frameworks. Struts2, Spring, Mule, and Wicket have already been integrated.

Quercus/PHP

- See Quercus/PHP documentation

Resin includes Quercus, our PHP 5 implementation, written entirely in Java. Quercus is a reliable PHP, compiling to Java code and taking advantage of the JDK's JIT compiler for maximum performance. Because Quercus is in Java, it's security is far superior to a C implementation. The JVM automatically protects Quercus from stack overruns, pointer overflows, and third party C-modules.

And the performance is great. The compiled Quercus code is 4 to 6 times faster than raw, mod_php code. Even when compared with PHP accelerators, Quercus code is equal or slightly faster, depending on the application.

Quercus integrates tightly with Java, naturally since it is written in Java. Resin-IoC applications can easily provide beans and components to a Quercus/PHP presentation layer through a simple `java_bean` call. For more involved integration, Quercus provides a complete API for generating PHP facades over Java libraries. All the PHP libraries are written to this public API.

With Quercus, Java sites have access to the PHP killer applications. For blogs, Quercus supports the extremely popular Wordpress. For wikis, Quercus supports MediaWiki, the engine behind Wikipedia.

Database Pooling

Resin provides a robust connection pool for any JDBC 1, JDBC 2, or JDBC 3 database driver. Resin's database pools integrate with full 2-phase XA transaction capabilities. In addition, the database connections can use round robin load balancing and backup failover for increased performance and reliability.

- See Resin databases documentation

EJB 3.0 (Enterprise Java Beans)

- See Resin EJB 3.0 documentation
- See Amber (JPA) documentation

Resin's EJB support is fully integrated with Resin-IoC and WebBeans. This integration means you can use only the EJB capabilities you need without the overhead of .ear packaging. (Although, .ears are fully supported.) A servlet, for example, can use EJB's @TransactionAttribute without any special packaging.

Server Push (Comet)

- See Resin Comet documentation.

As web users are expecting dynamic responses from web sites, the server push HTTP processing model has become important. Also called Comet, server push sends event or streaming data from the server to the client without waiting for a client poll.

Resin's Comet support focuses on solving the concurrency issues around server push. Since the servlet is now responding to events from internal services as well as managing HTTP data, the synchronization issues become more complex. The Resin Comet API is designed to cleanly separate the concurrency roles, letting the servlet continue to operate as a single-threaded request, while letting the event services send asynchronous messages.

JavaScript browsers and Flash applications are both supported by Resin's Comet. Flash can take advantage of Resin's Hessian support to stream binary-encoded events.

JSP 2.1 (Java Server Pages)

- See Resin JSP documentation

Resin's JSP implementation is tightly integrated with the JSTL and JSF implementations. Because of the tight binding, Resin can generate optimized JSP code, improving performance, reducing code size and reducing memory consumption.

Even JSP tag libraries can run faster with Resin's JSP. Resin bytecode-analyzes your tag libraries to determine which capabilities your actually using, and generates only that code necessary to run the features you're using.

The JSP implementation aggressively reuses your tag library instances, reducing memory consumption and improving performance. Because of Resin's reuse, application libraries must strictly follow the JSP tag specification.

Resin-IoC beans are always available through JSP EL expressions. This IoC integration means your custom components and beans are directly accessible to your HTML generating code.

JSTL 1.01 (Java Standard Template Library)

- See Resin JSTL documentation

Resin's JSTL works together with its JSP implementation to provide efficient, generated Java code. Since Resin's JSP is fully JSTL-aware, it can implement the JSTL tags directly in Java code, without involving the overhead of the JSTL tag libraries.

Servlet 2.5

- See Resin servlet documentation

Since the servlet engine of Resin is built on a Resin-IoC foundation, servlets can use WebBeans dependency injection, event observation, and interception. Remember that Resin-IoC is fully integrated with its EJB support? By transitivity, Resin servlets automatically have access to EJB transaction aspects without any extra overhead.

Remote services are tied into servlet capabilities as well. Because remote services are URL-based, they naturally fit into the the familiar servlet processing model. There's no need to introduce complexities like standard EJB deployment or add extra frameworks just to expose a service as a URL.

Resin Embedding

- See Resin embedding documentation

Resin embedding is a lightweight facade over the core Resin server suitable for automated testing frameworks and IDE environments. With Resin's embedded testing interface, your tests can send HTTP requests directly to the server,

skipping the overhead of TCP connections, and saving time on test setup and teardown.

The embedded API gives you hooks into Resin-IoC, so you can inject mock services into your tests, and still ensure that your application is tested in the full application server environment. Since integration testing uses the actual, deployment environment, elusive production bugs are easier to catch and destroy.

Resin Remoting

- See Resin remoting documentation

Resin remoting builds on the strengths of the Servlet architecture. As developers turn away from complicated remoting frameworks, the simplicity and elegance of the Servlet model becomes more appealing. Multithreaded, IoC-enabled, designed for HTTP and the web's URL architecture, well-known, well-documented, and well-integrated with application servers, the Servlet model has enjoyed phenomenal success.

Services expose remote APIs with the same interfaces and classes their clients will use, taking advantage of the Java compiler to enforce protocol compatiblity. Since Resin remoting protocols are available using a simple driver API based on Servlets, your services can choose the protocol to expose at deployment time, whether its a fast, binary wire-protocol like Hessian, or an XML-based wire-protocol like SOAP. Resin remoting clients select the wire-protocol matched with service API, and register with Resin-IoC/WebBeans, so application code can obtain the client proxy with simple, typed dependency injection.

JPA (Java Persistence Architecture)

- See Resin's Amber documentation

Resin Watchdog

- See Resin watchdog documentation

The Resin watchdog service manages each Resin instance, allowing for graceful recovery and restart from critical server deadlocks and crashes. As a management capability, it offers a centralized interface for starting, stopping, and checking the status of multiple JVM instances. Because the watchdog is well-placed to gather information, it is responsible for management and logging of the server's debugging or warning messages.

For unix systems, the watchdog also improves system security. While the watchdog will run as the root user, Resin JVM instances run as low-authorization users, protecting your system from application bugs and security holes.

As a side benefit of the watchdog, JVM parameters like heap settings, agent configuration, remote JMX and system properties can be configured in the

resin.xml itself. The watchdog can read the same resin.xml as the server itself, putting all configuration relevant to the server in one location.

For more complicated configurations like an ISP setting, the watchdog can use a separate configuration file readable only by root. The watchdog will launch Resin JVMs with assigned resin.xml files, ports, and user assignment, protecting each Resin JVM user from each other.

Messaging (JMS 1.1)

- See Resin messaging documentation

Resin's messaging focuses on simplifying the use and deployment of messaging applications. At the core, a messaging system is really just a queue with extra reliability and performance features like persistence, transactions, and clustered distribution. So Resin provides the standard queueing APIs like the JDK `BlockingQueue` as facades to the JMS API.

On the receiving end, `MessageListener` implementations are just Java services implementing a single `onMessage` method to receive the message, so there's no need for complications. The only configuration needed is a binding between the service class and the queue it wants to receive messages from. Three lines of XML is sufficient for that binding, with no need for any extra packaging.

Of course, the message listeners are can use all Resin-IoC capabilities, including injecting JPA `EntityManager` for database management, or other queues to filter and pass the message along.

Transactions (JTA 1.0.1b)

Resin's transaction manager handles full two-phase XA transaction suppport, including transaction logging for reliable transaction recovery. The transaction capabilities are integrated throughout Resin, so your databases, JCA connections, and EJBs work together. Since Resin's `UserTransaction` and `TransactionManager` are registered with Resin-IoC, injecting these managers into any IoC-enabled component is just adding an @In annotation.

Operating Systems

Resin and Resin-Professional are tested and supported on the following operating systems:

- Linux
- Solaris
- Windows
- Mac OS X

Resin is known to work on the following operating systems:

- AIX
- HP-UX
- Free-BSD

Resin is used on these operating systems, however there may or may not be difficulties with Apache integration, and the JNI code that Resin uses to increase performance may not be available.

Chapter 2

Installation

2.1 Resin Installation Quick Start

Quick Start for the Impatient

The Resin web server starts listening to HTTP requests on port 8080 and listens on port 6800 for load balancer or cluster messages. Resin can then be used for development or evaluation. The steps are:

1. Install JDK 1.5 or later.
   - On Unix, set the JAVA_HOME variable or link /usr/java to the java home.
   - On Windows, check to make sure the JDK installation sets JAVA_HOME correctly.
2. unzip/untar the Resin download. It will unzip into resin-3.2.x/.
   - resin-3.2.x is resin.home, the location of the Resin distribution.
   - For now, it is also ${resin.root}, the location of your content. Soon, you will want to move ${resin.root} to something like `/var/www` .
3. On Unix, execute use ./configure; make; make install
4. Start in development mode `java -jar resin-3.2.x/lib/resin.jar`
5. Browse `http://localhost:8080`

Adding Content

Once you've made sure Resin is working, you'll want to add some content to the default web site:

1. Add PHP files like `resin-3.2.0/hosts/default/webapps/ROOT/hello.php` .

- The URL in your browser will be `http://localhost:8080/hello.php`

2. Add JSP files like `resin-3.2.0/hosts/default/webapps/ROOT/hello.jsp` .
   - The URL in your browser will be `http://localhost:8080/hello.jsp`
3. Add servlets like `resin-3.2.0/hosts/default/webapps/ROOT/WEB-INF/classes/test/HelloSer`
   - Create a file `resin-3.2.0/hosts/default/webapps/hello/WEB-INF/resin-web.xml` to configure the servlet.
4. Add .war files like `resin-3.2.0/hosts/default/webapps/hello.war` .
   - The URL in your browser is `http://localhost:8080/hello`
5. Create web-apps directly like `resin-3.2.0/hosts/default/webapps/hello/index.php`
   The URL in your browser is `http://localhost:8080/hello` .
   - Create a file `resin-3.2.0/hosts/default/webapps/hello/WEB-INF/resin-web.xml` to configure the 'hello' web application.

Virtual Hosts

You can easily create virtual hosts by creating content in the `resin.root/hosts` directory:

1. Add a hello.php to `resin-3.2.0/hosts/localhost/webapps/ROOT/hello.php`

Permanent content locations

Eventually, you'll want to move your content and configuration into a more permanent location:

1. Create a permanent resin.root:
   - Virtual hosts go in `/var/www/hosts/www.foo.com/webapps/ROOT`
   - Run `java -jar resin-3.2.0/lib/resin.jar --resin-root /var/www`
   - You can also set <root-directory> in the <cluster> to configure the resin root.
2. If needed, modify the Resin configuration in `resin-3.2.0/conf/resin.xml`
   - You can copy resin.xml to somewhere like `/etc/resin/resin.xml`
   - Run `java -jar resin-3.2.0/lib/resin.jar --conf /etc/resin/resin.xml`

Running Resin as a daemon

In a deployment environment, Resin will run as a background daemon. The previous steps ran Resin in the foreground, which is convenient for development since the logging output goes to the console. When running as a daemon, Resin detaches from the console and continues running until told to stop.

1. Start resin with `java -jar resin-3.2.0/lib/resin.jar start`
2. Stop resin with `java -jar resin-3.2.0/lib/resin.jar stop`
3. Restart resin with `java -jar resin-3.2.0/lib/resin.jar restart`

Until you're ready to deploy the server, those are all the steps needed to get started with Resin.

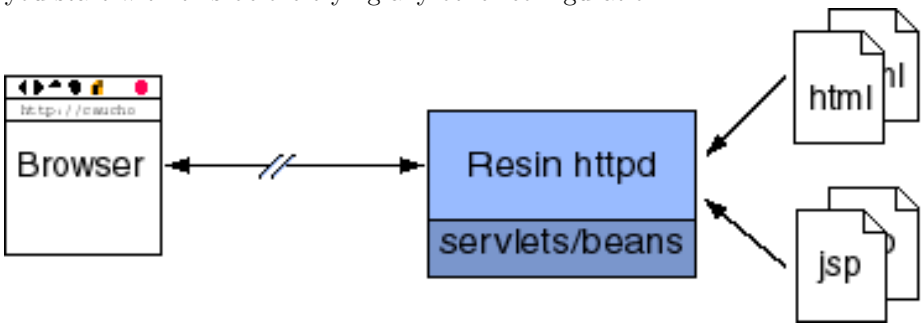
Preconditions

Resin 3.2 needs Java before it can run. It needs JDK 1.5 or a later JDK.

Sun's JDK for Windows, Solaris, and Linux can be found at http://java.sun.com. Sun also has links to some other ports of the JDK.

Resin Web Server

The easiest and fastest Resin configuration uses the Resin as the primary or only web server. This configuration provides a Java HTTP server. We recommend you start with this before trying any other configuration.



The server listens at port 8080 in the default configuration and can be changed to the HTTP port 80 during deployment.

Windows

1. Install JDK 1.5 or later.
2. Check that the environemnt variable JAVA_HOME is set to the JDK location, e.g. "c:\j2sdk1.5.0_01"
3. Unzip resin-3.2.0.zip

4. Define the environment variable RESIN_HOME to the location of Resin, for example "c:\resin-3.2.0"
5. Execute `java -jar resin-3.2.0/lib/resin.jar`
6. Browse http://localhost:8080

Unix (including MacOS-X)

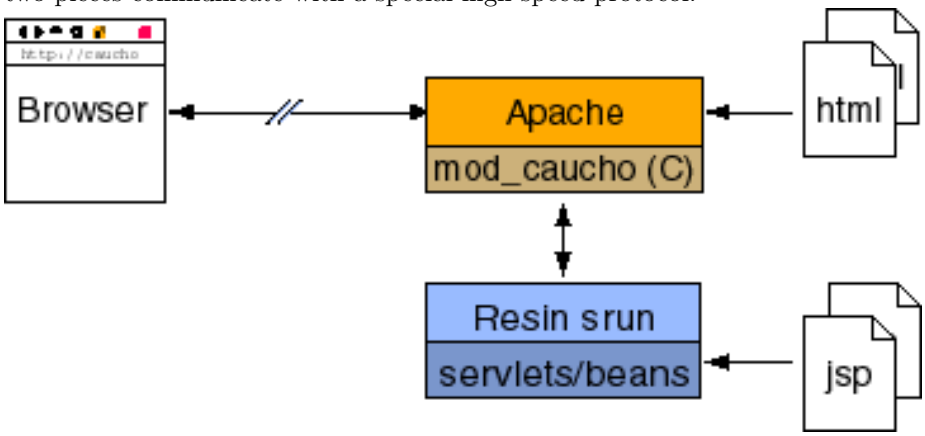
1. Install JDK 1.5 or later and link /usr/java to the Java home or define the environment variable JAVA_HOME.
2. tar -vzxf resin-3.2.0.tar.gz
3. cd resin-3.2.0
4. ./configure
5. make
6. make install
7. Execute `java -jar resin-3.2.0/lib/resin.jar`
8. Browse http://localhost:8080

For more details, see the Resin Web Server configuration page.

Resin with Apache

If you are already using Apache for your web server, you can use Resin with Apache. This configuration uses Apache to serve html, images, PHP, or Perl, and Resin to serve JSPs and Servlets.

The Apache configuration uses two pieces: a C program extending Apache (`mod_caucho`) and Java program supporting servlets and JSP (`srun` .) The two pieces communicate with a special high-speed protocol.



To configure Apache with Resin, you must configure both Apache and Resin. The Resin configuration is identical to Resin's httpd configuration. The Apache configuration tells Apache how to find Resin.

1. On Unix only, compile mod_caucho.so using `./configure --with-apache; make`
2. Make any needed Apache httpd.conf changes
3. Make any needed Resin resin.xml changes
4. Restart Apache
5. Start Resin with `resin-3.2.0/bin/resin.sh` on Unix or `resin-3.2.0/resin.exe` on Windows.

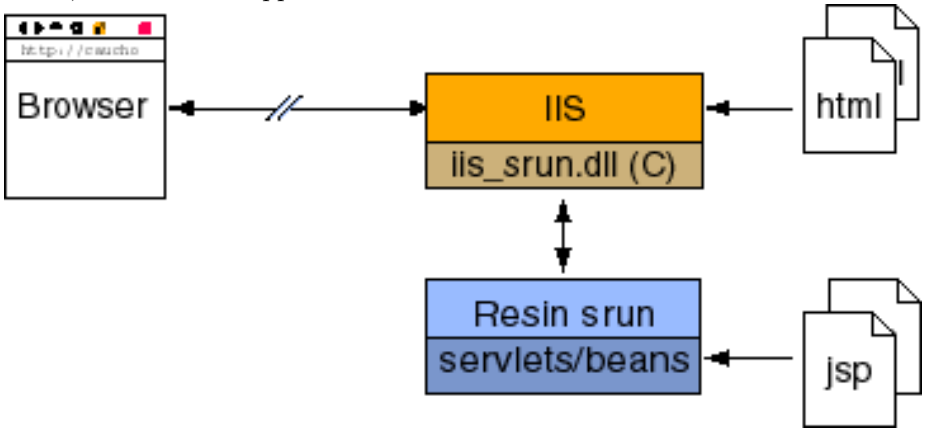
On Unix, you'll run configure using `--with-apache` and then `make` :

```
unix> ./configure --with-apache=/usr/local/apache
unix> make
unix> make install
```

For more details, see the Resin with Apache configuration page.

Resin with IIS

You can also combine IIS and Resin. IIS serves static content like html and images and Resin serves JSPs and Servlets. The IIS configuration requires two pieces: `isapi_srun.dll` , an ISAPI extension which lets IIS talk to Resin, and `srun` , Resin's Java support.



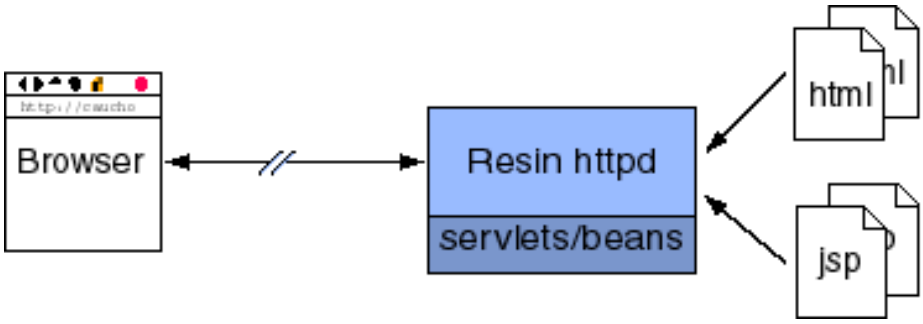
For this setup you must configure both IIS and Resin. The Resin configuration to run with IIS is identical to Resin's standalone configuration. The IIS configuration tells IIS how to find Resin.

1. Setup the registry and IIS using resin-3.2.0/setup.exe
2. Any needed Resin resin.xml changes
3. Restart IIS
4. Start Resin with resin-3.2.0/resin.exe.

For more details and troubleshooting steps, see the Resin with IIS configuration page.

2.2 Resin Installation

2.3 Resin Web Server



Unix (including Linux and MacOS-X)

Getting Started

The following steps will start Resin for development:

1. Install JDK 1.5 or later and link `/usr/java` to your Java home or set environment variable `JAVA_HOME` .
2. `tar -vzxf resin-3.2.0.tar.gz` in `/usr/local/share`
3. (Optional) Link `/usr/local/share/resin` to the `resin-3.2.0` directory.
4. ./configure; make; make install
   - some ./configure options are available
5. Run `java -jar resin/lib/resin.jar`
   - Or run `resin/bin/resin.sh`
6. Browse `http://localhost:8080`

```
unix> java -jar resin/lib/resin.jar
Resin Professional 3.2.0 (built Wed, 06 Aug 2008 12:49:30 PDT)
Copyright(c) 1998-2008 Caucho Technology.  All rights reserved.

  001111.license - 1 Resin server Hogwarts School

Starting Resin on Thu, 07 Aug 2008 19:11:52 -0700 (PDT)

[21:22:21.282] Proxy Cache disk-size=1024M memory-size=8M
[21:22:21.477] Server[id=,cluster=app-tier] starting
[21:22:21.477]
[21:22:21.477] Mac OS X 10.4.7 ppc
[21:22:21.477] Java 1.5.0_06-64, 32, mixed mode, sharing, MacRoman, en, "Apple Computer, Inc."
[21:22:21.477] resin.home = /usr/local/share/resin
[21:22:21.478] resin.root = /var/www
[21:22:21.478]
[21:22:21.492] Loaded Socket JNI library.
[21:22:21.595] hmux listening to localhost:6800
[21:22:21.696] http listening to *:8080
[21:22:21.842] Host[] starting
[21:22:22.089] WebApp[] starting
```

Successful Foreground Startup Output

Deployment Directories

When deploying, it's a good idea to create a bit of structure to make Resin and website upgrades easier and more maintainable.

1. Create a user to run Resin (e.g. `resin` or another non-root user)
2. Link `/usr/local/share/resin` to the current Resin directory. This is `$RESIN_HOME` .
3. Create a deployment root, e.g. `/var/www` , owned by the resin user. This is `$RESIN_ROOT` .
4. Put the modified `resin.xml` in `/etc/resin/resin.conf`
5. Put the site documents in `/var/www/hosts/default/webapps/ROOT` .
6. Put any .war files in `/var/www/hosts/default/webapps` .
7. Put any virtual hosts in `/var/www/hosts/www.foo.com` .
8. Output logs will appear in `/var/www/log` .
9. Create a startup script and configure the server to start it when the machine reboots.

Startup Script

You can create your own startup script which will start and stop the Resin-Watchdog, and will pass any command-line arguments. The script might typically do a number of things:

1. Configure the location of Java in `JAVA_HOME`
2. Configure the location of Resin in `RESIN_HOME`
3. Configure your web site directory in `RESIN_ROOT`
4. Select a server and pid file if you have multiple Resin servers.
5. Start and stop the ResinWatchdog.

The start script might look like:

```
#!/bin/sh

JAVA_HOME=/usr/java
RESIN_HOME=/usr/local/share/resin
RESIN_ROOT=/var/www

java=$JAVA_HOME/bin/java

export JAVA_HOME
export RESIN_HOME
export RESIN_ROOT

$java -jar $RESIN_HOME/lib/resin.jar \
      -resin-root $RESIN_ROOT \
      -conf /etc/resin/resin.xml \
      -server a \
       $*
```

Example start.sh script

This script would be called as `./start.sh start` to start and `./start.sh stop` to stop.

The `-server` argument is only necessary if you have multiple servers (JVM instances) either on different machines or the same machine. The load balancing and distributed sessions pages describe when you might use `-server` .

More information on deploying on Unix is available on the Linux boot documentation.

Windows

Getting Started

1. Install JDK 1.5 or later.

2. Make sure the JDK installation set the environment variable `JAVA_HOME` correctly

3. Unzip resin-3.2.0.zip

4. Run `java -jar resin-3.2.0/lib/resin.jar`

5. Or execute `resin-3.2.0/resin.exe`

6. Browse `http://localhost:8080`

```
C:\win32> resin-3.2.0\bin\resin.exe
Resin 3.2.0 (built Wed Aug 06 18:21:13 PST 2008)
Copyright(c) 1998-2008 Caucho Technology.  All rights reserved.

Starting Resin on Thu, 07 Aug 2008 19:11:52 -0500 (EST)
[19:11:56.479] ServletServer[] starting
[19:11:57.000] Host[] starting
[19:11:58.312] Application[http://localhost:8380/doc] starting
[19:12:11.872] Application[http://localhost:8380/quercus] starting

...

[19:12:12.803]http listening to *:8380
[19:12:12.933]hmux listening to *:6802
```

Starting on Win32

Deploying as a Windows Service

The Resin Web Server can be installed as an Windows service.

To install the service, use

```
C:\> resin-3.2.x\resin.exe -install -conf conf/myconf.xml \
                              -user MyResinUser -password mypassword
```

To remove the service, use

```
C:\> resin-3.2.x\resin.exe -remove
```

You will either need to reboot the machine or start the service from the ControlPanel/Services panel to start the server. On a machine reboot, Windows will automatically start the web server.

You can also start and stop the service from the command-line:

```
C:\> net start resin
...
C:\> net stop resin
```

Resin's `-install` saves the command-line arguments and starts the service with those arguments. You can look view them in the control panel, under the executable string.

With multiple named servers, you can use `-install-as foo` to specify the service name.

```
C:\> resin-3.2.x\resin.exe -install-as ResinA \
                           -conf conf/myconf.conf \
                           -server a
C:\> net start ResinA
```

Running Resin

Processes Overview

Resin runs as multiple processes that begin with the following JVM command:

```
unix> java -jar /usr/local/share/resin/lib/resin.jar \
           -conf /etc/resin/resin.xml \
           start
```

The `-jar` argument tells java to run the Main-Class defined in resin.jar's manifest. The `-conf` argument specifies the path to your Resin configuration file. Lastly, Resin accepts `start` , `stop` , and `restart` arguments which are passed to the watchdog process. An additional command-line option, `-server` is used in load-balanced deployments.

JDK 1.5 includes a jps command which will show the pids of any java processes.

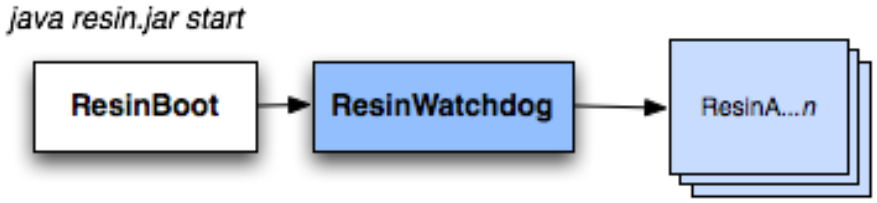
```
unix> jps
2098 Jps
2064 ResinWatchdogManager
2097 Resin
```

Example jps Process List

When running as a daemon (eg, `resin.sh start`) `ResinWatchdogManager` is the watchdog and `Resin` is the actual Resin instance. When running Resin

as a foreground process, the process list displays `resin.jar` , which acts as the watchdog.

The first process that starts is the actual startup program, `java -jar resin.jar` . It passes command-line arguments to the second process, the ResinWatchdogManager. This watchdog process takes care of starting the actual Resin process(es). ResinWatchdogManager monitors the state of Resin and restarts it if necessary, improving reliability.



The Watchdog Process

The ResinWatchdogManager is the parent process, providing automatic restarting Resin in cases of failure, and providing a single point of control for the `start` , `stop` and `restart` of all Resin processes. It is responsible for launching Resin with the correct JVM arguments and environment options such as starting Resin as the specified user, e.g. for <user-name> on unix.

ResinWatchdogManager watches Resin via a Socket connection. When the watchdog socket closes, Resin shuts itself down gracefully. The watchdog closes the socket on a `stop` or `restart` or if the watchdog itself is killed. If Resin exits for any reason, the watchdog will automatically start a new Resin process. This socket connection approach avoids the need for any signals or actual killing of Resin from the watchdog, and also makes it easy to stop all the Resins if necessary by just killing the watchdog.

The ResinWatchdogManager doesn't actually kill Resin or even check Resin's status, it just checks to see if Resin is alive or not. So if the JVM were to completely lock up, the watchdog would still think Resin was okay and would take no action.

Resin Processes

If Resin detects a major error (like running out of memory) or if the resin.xml changes, it will exit and the watchdog would start a new Resin instance. Reasons a Resin instance might exit include:

- resin.xml changes
- out of memory error
- detected deadlocks

- segv and other severe errors

Because the watchdog is always managing Resin processes, if you ever need to stop Resin with `kill` , you must kill the watchdog. Just killing the Resin process results in the watchdog restarting it automatically.

Logging

The watchdog will log to `log/watchdog-manager.log` . The Resin standard out/err is `log/jvm-servername.log` . ResinWatchdogManager is responsible for creating both of these log files, so `jvm-servername.log` is not really under the control of the Resin instance. This makes it somewhat more reliable in case of JVM deadlocks, etc.

2.4 Resin with Apache

If you have not yet done so, we suggest you use the Resin standalone web server option first.

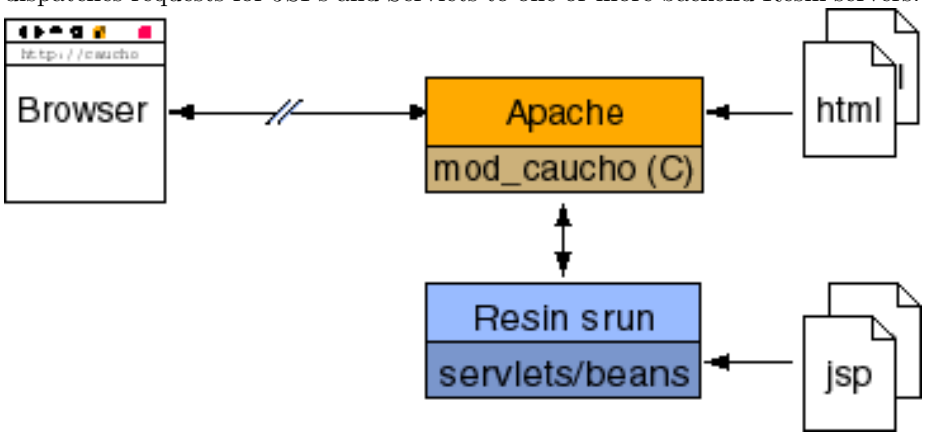
Before you integrate Resin with Apache

Before integrating Resin with Apache, it is valuable to configure Resin as a standalone server, especially with more complicated setups such as those involving virtual hosts. Doing so isolates the steps and makes troubleshooting easier.

Many users find that the performance, flexibility, and features of Resin make Resin a desirable replacement for Apache.

How Resin integrates with Apache

When used with Apache, Resin serves JSPs and Servlets and Apache serves static content like html and images. Apache is a frontend server, it handles the request from the browser. Resin's `mod_caucho` plugin integrates with Apache, it dispatches requests for JSPs and Servlets to one or more backend Resin servers.



mod_caucho queries the backend server to distinguish the URLs going to Resin from the URLs handled by Apache. The backend server uses the <servlet-mapping> directives to decide which URLs to send. Also, any *.war file automatically gets all its URLs. Other URLs stay with Apache.

There's a more complete discussion of the URL dispatching in the How the Plugins Dispatch to Resin page.

Unix Installation

Resin needs Apache 1.3.x or greater and DSO support.

To configure Resin with Apache, you must follow the following steps:

1. Compile Apache
2. Compile mod_caucho.so
3. Configure Apache
4. Set up environment
5. Configure resin.xml
6. Restart Apache and start the backend Resin server

Compiling Apache

You need a version of Apache with DSO support enabled. Apache has full documentation at their website. To check if your apache has DSO support, you can check for `mod_so.c` in your your httpd

```
unix> /usr/local/apache/bin/httpd -l
Compiled-in modules:
  ...
  mod_so.c
  ...
```

checking apache httpd for mod_so.c

Many distributions, e.g. Red Hat Linux, will have Apache preinstalled. However, because the standard distribution has files all over the place, some people prefer to recompile Apache from scratch.

Once you untar Apache, build it like:

```
unix> ./configure --prefix=/usr/local/apache --enable-module=so
unix> make
unix> make install
```

Solaris versions of Apache may need additional flags, otherwise you'll get some linking errors when trying to load Resin. You may need to refer to the Apache documentation if you get linking errors. Here's an example configuration on Solaris:

```
unix> ./configure --prefix=/usr/local/apache \
                  --enable-rule=SHARED_CORE \
                  --enable-rule=SHARED_CHAIN \
                  --enable-module=so \
                  --enable-module=most \
                  --enable-shared=max
```

Compiling mod_caucho.so

To compile and install `mod_caucho` on Unix, you'll need to run Resin's `configure` and then make. This step will create mod_caucho.so and put it in the Apache module directory. Usually, mod_caucho.so will end up in /usr/local/apache/libexec/mod_caucho.so.

If you know where your `apxs` executable is, you can use –with-apxs. `apxs` is a little Perl script that the Apache configuration makes. It lets modules like Resin know how all the Apache directories are configured. It is generally in /usr/local/apache/bin/apxs or /usr/sbin/apxs. It's usually easiest to use –with-apxs so you don't need to worry where all the Apache directories are.

```
unix> ./configure --with-apxs=/usr/local/apache/bin/apxs
unix> make
```

Even if you don't know where `apxs` is, the configure script can often find it:

```
unix> ./configure --with-apxs
unix> make
```

As an alternative to –with-apxs, if you've compiled Apache yourself, or if you have a simple configuration, you can generally just point to the Apache directory:

```
unix> ./configure --with-apache=/usr/local/apache
unix> make
unix> make install
```

The previous –with-apxs or –with-apache should cover most configurations. For some unusual configurations, you can have finer control over each directory with the following arguments to ./configure. In general, you should use –with-apache or –with-apxs, but the other variables are there if you know what you're doing.

–with-apache=dir	The Apache root directory.
–with-apxs=apxs	Pointer to the Apache extension script
–with-apache-include=dir	The Apache include directory
–with-apache-libexec=dir	The Apache module directory
–with-apache-conf=httpd.conf	The Apache config file

Configure the Environment

If you don't already have Java installed, you'll need to download a JDK and set some environment variables.

Here's a typical environment that you might put in /.profile or /etc/profile

```
# Java Location
JAVA_HOME=/<installdir>/jdk1.4
export JAVA_HOME

# Resin location (optional).  Usually Resin can figure this out.
RESIN_HOME=/<installdir>/resin-3.2.2
export RESIN_HOME

# If you're using additional class libraries, you'll need to put them
# in the classpath.
CLASSPATH=
```

Windows Installation

The `setup.exe` program installs the mod_caucho.dll plugin for any Apache it finds, and modifies the Apache httpd.conf file.

The httpd.conf file is also easily modified manually:

```
LoadModule caucho_module \
    <installdir>/resin-3.2.x/win32/apache-2.0/mod_caucho.dll

ResinConfigServer localhost 6800
<Location /caucho-status>
  SetHandler caucho-status
</Location>
```

httpd.conf

Configuring resin.xml

The communication between `mod_caucho` and the backend Resin server takes place using a `server` port.

The `resin.xml` for the backend server contains a `server` to enable the port. The default `resin.xml` has an server listener on port 6800.

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">

  ...

  <cluster id="app-tier">

    ...

    \textbf{\ensuremath{<}server id="" address="127.0.0.1" port="6800"/\ensuremath{>}}

    ...
```

resin.xml

The resin.xml and the layout of your webapps should match the layout that Apache expects. The mapping of urls to filesystem locations should be consistent between Apache and the backend Resin server.

The default resin.xml looks in `resin-3.2.x/webapps/ROOT` for JSP files and `resin-3.2.x/webapps/ROOT/WEB-INF/classes` for servlets and java source files. To tell Resin to use Apache's document area, you configure an explicit `web-app` with the appropriate document-directory:

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">

  ...

  <server>

    ...

    <host id="">

      <web-app id='/' document-directory="/usr/local/apache/htdocs"/>

    </host>

    ...

  </server>
</resin>
```

resin.xml

Starting the app-tier Resin server

Now you need to start the app-tier Resin server. Starting Resin is the same with Apache or standalone. See the Resin Web Server page for a detailed description.

```
unix> java -jar $RESIN_HOME/lib/resin.jar

unix> bin/resin.sh

win> resin.exe
```

```
Resin 3.2.2 (built Mon Aug  4 09:26:44 PDT 2006)
Copyright(c) 1998-2006 Caucho Technology.  All rights reserved.

Starting Resin on Mon, 04 Aug 2006 09:43:39 -0700 (PDT)
[09:43:40.664] Loaded Socket JNI library.
[09:43:40.664] http listening to *:8080
[09:43:40.664] ServletServer[] starting
[09:43:40.879] hmux listening to *:6800
[09:43:41.073] Host[] starting
[09:43:41.446] Application[http://localhost:8080/resin-doc] starting
[09:43:41.496] Application[http://localhost:8080] starting
```

Resin will print every port it's listening to. In the above example, Resin has an http listener on port 8080 and an server listener on port 6800 (using its custom 'hmux' protocol). mod_caucho establishes connections to Resin using

port 6800, and a web browser can connect using port 8080. Usually the 8080 port will be unused, because web browsers will make requests to Apache, these requests get dispatched to Resin as needed by mod_caucho. A Resin configured http listener on port 8080 is a useful debugging tool, it allows you to bypass Apache and make a request straight to Resin.

The following snippet shows the <http> and <server> configuration for the above example.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">
    <server-default>
      <http address="*" port="8080"/>
    </server-default>

    <server id="" address="192.168.2.10" port="6800"/>

    ...
  </cluster>
</resin>
```

Testing the servlet engine

Create a test file '/usr/local/apache/htdocs/test.jsp'

```
2 + 2 = <%= 2 + 2 %>
```

Browse http://localhost/test.jsp again. You should now get

```
2 + 2 = 4
```

Configuring Apache httpd.conf

The installation process above automatically changes the httpd.conf file. You can also configure the httpd.conf file manually, or modify the default configuration created by the installation process.

```
LoadModule caucho_module libexec/mod_caucho.so

ResinConfigServer localhost 6800
<Location /caucho-status>
  SetHandler caucho-status
</Location>
```

Unix - httpd.conf

```
LoadModule caucho_module \
    <installdir>/resin-3.2.x/libexec/apache-2.0/mod_caucho.dll

ResinConfigServer localhost 6800
<Location /caucho-status>
  SetHandler caucho-status
</Location>
```

Windows - httpd.conf

The `ResinConfigServer` is used to tell mod_caucho how to contact the backend Resin server. The backend Resin server tell's mod_caucho which urls should be dispatched.

APACHE COMMAND	MEANING
ResinConfigServer `host port`	Specifies the Resin JVM at `host:port` as a configuration server.

caucho-status

`caucho-status` is optional and probably should be avoided in a production site. It lets you ask the Caucho Apache module about it's configuration, and the status of the backend server(s), valuable for debugging.

After any change to httpd.conf, restart Apache. Now browse http://localhost/caucho-status.

Manual configuration of dispatching

You can also dispatch to Resin directly from the httpd.conf. Instead of relying on the ResinConfigServer directive to determine which url's to dispatch to the backend server, Apache handler's are used to specify the url's to dispatch.

```
CauchoHost 127.0.0.1 6800

<Location /foo/*>
  SetHandler caucho-request
</Location>
```

APACHE COMMAND	MEANING
CauchoHost `host port`	Alternative to `ResinConfigServer` , adds the Resin JVM with an server port at host:port as a backend server.

CauchoBackup `host port`	Alternative to `ResinConfigServer` , adds the Resin JVM with a server port at host:port as a backup back-end server.

APACHE HANDLER	**MEANING**
caucho-status	Handler to display /caucho-status
caucho-request	Dispatch a request to Resin

Requests dispatched directly from the Apache httpd.conf will not appear in /caucho-status.

Virtual Hosts

The virtual host topic describes virtual hosts in detail. If you're using a single JVM, you only need to configure the resin.xml.

```
LoadModule caucho_module libexec/mod_caucho.so

ResinConfigServer 192.168.0.1 6800
<Location /caucho-status>
  SetHandler caucho-status
</Location>
```

httpd.conf

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="app-tier">

  <server id="" address="192.168.0.1" port="6800"/>

  <host id='www.gryffindor.com'>
    <host-alias>gryffindor.com</host-alias>
     ...
  </host>

  <host id='www.slytherin.com'>
    <host-alias>slytherin.com</host-alias>
     ...
  </host>
</cluster>
</resin>
```

resin.xml

Virtual Host per JVM

If you want a different JVM for each virtual host, your httpd.conf can specify a different server port for each host.

```
<VirtualHost gryffindor.com>
ServerName gryffindor.com
ServerAlias www.gryffindor.com
ResinConfigServer 192.168.0.1 6800
</VirtualHost>

<VirtualHost slytherin.com>
ServerName slytherin.com
ServerAlias www.slytherin.com
ResinConfigServer 192.168.0.1 6801
</VirtualHost>
```

httpd.conf

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

  <server id="" address="192.168.0.1" port="6800"/>

  <host id="">
     ...
  </host>
</cluster>
</resin>
```

gryffindor.conf

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster>

  <server id="" address="192.168.0.1" port="6801"/>

  <host id="">
     ...
  </host>
</cluster>
</resin>
```

slytherin.conf

```
$ bin/resin.sh -pid gryffindor.pid -conf conf/gryffindor.conf start
$ bin/resin.sh -pid slytherin.pid -conf conf/slytherin.conf start

...

$ bin/resin.sh -pid gryffindor.pid stop
```

Load Balancing

The Reliability and Load Balancing section provides an introduction to the concepts of load balancing.

mod_caucho recognizes cluster configurations for load balancing. Requests are distributed to all machines in the cluster, all requests in a session will go to the same host, and if one host goes down, Resin will send the request to the next available machine. Optional backup machines only receive requests if all of the primaries are down.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">
    <server id="a" address="192.168.0.11" port="6800" index="1"/>
    <server id="b" address="192.168.0.11" port="6801" index="2"
            backup="true"/>
    <server id="c" address="192.168.0.12" port="6800" index="3"/>
    <server id="d" address="192.168.0.12" port="6801" index="4"
            backup="true"/>
    ...
  </cluster>
</resin>
```

resin.xml

mod_caucho only needs to know about one of the backend servers. It will query that backend server, and learn about all of the other members of the cluster.

```
ResinConfigServer 192.168.0.11 6800
```

mod_caucho keeps a local cache of the configuration information, so if the backend server becomes unavailable then the cached configuration will be used until the backend server becomes available again.

The httpd.conf file can also specify more than one backend server, when mod_caucho checks for configuration updates, it will check each in turn, and only if none of them are available will it use the local cached copy.

```
ResinConfigServer 192.168.0.11 6800
ResinConfigServer 192.168.0.12 6801
```

Manual configuration of load balanced dispatching

Manual dispatching in httpd.conf can also specify the backend hosts and the backend backup hosts, as an alternative to using `ResinConfigServer` .

```
CauchoHost 192.168.0.11 6800
CauchoBackup 192.168.0.11 6801
CauchoHost 192.168.0.12 6800
CauchoBackup 192.168.0.12 6801

<Location /foo/*>
  SetHandler caucho-request
</Location>
```

Manual configuration of location based dispatching

```
<Location /applicationA/*>
  ResinConfigServer 192.168.0.11 6800
</Location>

<Location /applicationB/*>
  ResinConfigServer 192.168.0.12 6800
</Location>
```

Troubleshooting

1. First, check your configuration with Resin standalone.sh. In other words, add a <http port='8080'/> and check port 8080.

2. Check http://localhost/caucho-status. That will tell if mod_caucho has properly connected to the backend Resin server.

3. Each server should be green and the mappings should match your resin.xml.

4. If caucho-status fails entirely, the problem is in the mod_caucho installation and the Apache httpd.conf.

5. If caucho-status shows the wrong mappings, there's something wrong with the resin.xml or the pointer to the backend server in httpd.conf.

6. If caucho-status shows a red servlet runner, then Resin hasn't properly started.

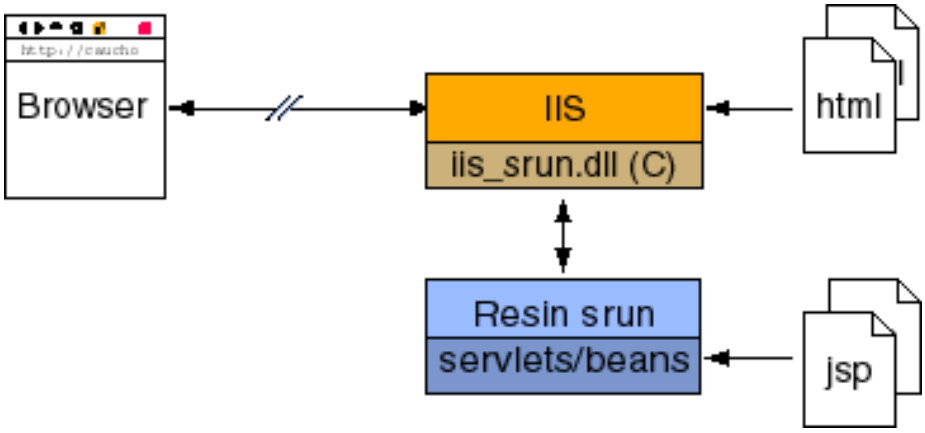
7. If you get a "cannot connect to servlet engine", caucho-status will show red, and Resin hasn't started properly.

8. If Resin doesn't start properly, you should look at the logs in resin-3.2.x/log. You should start `resin.sh -verbose` or `resin.exe -verbose` to get more information.

9. If Resin never shows a "hmux listening to *:6800" line, it's not listening for connections from mod_caucho. You'll need to add a <server> line.

10. If you get Resin's "file not found", the Apache configuration is good but the resin.xml probably points to the wrong directories.

2.5 Resin with IIS



If you have not yet done so, we suggest you consider the Resin web server option first, either as a standalone web server or as a web-tier server load-balancing to Resin app-tier servers. Resin's Java web server provides management, clustering, and load-balancing with failover.

Prerequisites and Environment Variables

Resin requires a 1.5 JDK. You can download one from Sun and install it.

The Control Panel is used to configure two environment variables:

```
JAVA_HOME=C:\jdk1.5.0
RESIN_HOME=C:\resin-3.2
```

Configuring IIS/PWS

To configure Resin with IIS, you must follow the following steps:

1. Configure IIS/PWS
2. Configure resin.xml
3. Start resin.exe

ISAPI Filter

You should run `RESIN_HOME/setup.exe` to setup your configuration. If setup.exe is not used, or it fails, the steps in Manual Configuration are necessary.

ISAPI Filter Priority

isapi_srun.dll installs itself as the default priority. Some users may need to set the priority to a higher level, e.g. to override IIS's DAV support.

```
ResinConfigServer localhost 6802
CauchoStatus yes
IISPriority high
```

resin.ini

Configuring resin.xml

resin.xml should mirror the configuration of IIS. In other words, you need to configure the document root and any directory aliases.

For many users, the only need to change needed in Resin is to change the attribute from 'webapps/ROOT' to something like 'C:/inetpub/wwwroot'. The mapping of url paths from the browser to real files on the disk must be the same for Resin as they are for IIS. For more complicated configurations that use mappings in IIS, you'll need to add path-mapping attributes to match.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

  <!-- configures the default host, matching any host name -->
  <host id="">

  <!-- configures the root web-app -->
  <web-app id='/'>
    <root-directory>C:/inetpub/wwwroot</root-directory>
    <!-- adds xsl to the search path -->
    <class-loader>
      <simple-loader path="$host-root/xsl"/>
    </class-loader>
  </web-app>
</host>

</cluster>
</cluster>
```

Example: resin.xml

Testing the servlet engine

From a `cmd` shell run `RESIN_HOME/resin.exe` to start the servlet runner.

```
C:\> cd %RESIN_HOME%
C:\resin-3.2> resin.exe
```

Run resin.exe

Now browse http://localhost/test.jsp. You should get a 'file not found' message.

Create a test file 'd:\inetpub\wwwroot\test.jsp'

```
2 + 2 = <%= 2 + 2 %>
```

Browse http://localhost/test.jsp again. You should now get

```
2 + 2 = 4
```

Command line arguments

The following configuration line arguments are recognized by resin.exe. When command line arguments are used along with `-install` to install as a service, the arguments are used each time the service starts.

ARGUMENT	MEANING
-verbose	Write more verbose information to the log file
-resin_home <path>	Sets the location of Resin
-java_home <path>	Specify the JDK location
-msjava	Use Microsoft's JVM
-nojit	Disable JIT compilation to help debugging
-classpath <cp>	Add to the classpath
-J<arg>	Set a Java command line argument, e.g. -J-nojit.
-X<arg>	Set a Java -X command line argument, e.g. -Xms128m.
-D<foo=bar>	Set a Java variable, e.g. -Dresin.home=here.
-install	Install as an NT service
-install-as <name>	Install as an NT service with the specific name.
-remove	Remove as an NT service
-remove-as <name>	Remove as an NT service with the specific name.

Memory Configuration

Memory configuration is part of the JVM's arguments. For most JVMs, you can find the full list by starting "java -X".

ARGUMENT	MEANING
-Xms<size>	Initial Java heap size, e.g. -Xms32m
-Xmx<size>	Maximum Java heap size, e.g. -Xmx128m
-Xms<size>	the size of the heap for the young generation

Memory Configuration

See Performance : JVM Tuning : Memory for more information on JVM memory usage.

Deploying as an NT Service

Once you're comfortable with using Resin with IIS, you can install it as an NT service. As a service, Resin will automatically start when NT reboots. The service will also automatically restart Resin if it unexpectedly exits.

To install the service, use

```
win> resin-3.2.x/resin.exe -install
```

To remove the service, use

```
win> resin-3.2.x/resin.exe -remove
```

You will either need to reboot the machine or start the service from the ControlPanel/Services panel to start the server. On a machine reboot, NT will automatically start the servlet runner.

Note: There is a bug in many JDKs which cause the JDK to exit when the administrator logs out. JDK 1.4 and later can avoid that bug if the JDK is started with -Xrs.

```
win> resin-3.2.x/resin.exe -install -Xrs
```

Load Balancing

With Resin, you can distribute requests to multiple machines. All requests with the same session will go to the same host. In addition, if one host goes down, the IIS filter will send the request to the next available machine.

In addition, you can specify backup machines. The backup only will serve requests if all primaries are down.

See the Resin config section for more details.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="app-tier">

    <server id="a" address="host1" port="6802" index="1"/>
    <server id="b" address="host2" port="6802" index="2"/>
    <server id="c" address="backup" port="6802" index="3" backup="true"/>
  ...
</cluster>
</resin>
```

resin.xml

```
win> resin-3.2.x/resin.exe -install-as "Resin-A" -server a -Xrs
win> resin-3.2.x/resin.exe -install-as "Resin-B" -server b -Xrs
win> resin-3.2.x/resin.exe -install-as "Resin-C" -server c -Xrs
```

installing multiple servlet runners as services

Manual Configuration

Experts may want to configure Resin/IIS by hand instead of using the setup program. The steps involved are:

1. Make sure resin.exe works
2. Copy isapi_srun.dll to the IIS scripts directory, `d:\inetpub\scripts` . You may need to run `net stop w3svc` to get permission to overwrite the file.
3. If you have a virtual site (virtual hosts), you must configure IIS to have the virtual directory `/scripts` point to `d:\inetpub\scripts` for each virtual site.
4. (optional) Create a resin.ini in `d:\inetpub\scripts` pointing to the ResinConfigServer
5. (optional) Add a "CauchoStatus yes" line to the resin.ini for debugging
6. Configure IIS to load isapi_srun.dll as an ISAPI filter.
7. Restart IIS (control panel/services) or `net stop w3svc` followed by `net start w3svc` .
8. Browse /servlet/Hello and /foo.jsp. You should see a "cannot connect" error.
9. Start resin.exe
10. Browse /servlet/Hello and /foo.jsp. You should now see the servlet.

Copying isapi_srun.dll to `inetpub/scripts` directory is relatively straightforward. If you're upgrading to a new version of Resin, you may need to stop IIS (control panel/services) to get permission to overwrite isapi_srun.dll.

The resin.ini is an optional file in `inetpub/scripts` to override the automatic registry $RESIN_HOME/conf/resin.xml configuration file. If you only have one Resin server, you should not create a resin.ini and let isapi_srun.dll use the registry value set by the setup.exe program.

resin.ini is only needed if you have multiple Resin configuration files for different IIS virtual hosts.

The resin.ini should contain the following line:

```
ResinConfigServer localhost 6802
```

You can change the host from `localhost` to a backend server. You can also add multiple `ResinConfigServer` items to cluster the configuration.

For debugging, you can add a "CauchoStatus yes" line to the resin.ini:

```
ResinConfigServer localhost 6802
CauchoStatus yes
```

For security purposes, the default value of CauchoStatus is "no" when you have a resin.ini.

Adding an ISAPI filter is accomplished in the IIS manager.

IIS and Resin on different machines

When Resin and IIS are on different machines, you'll change the ResinConfigServer from "localhost" to the IP address of the Resin server.

```
ResinConfigServer 192.168.0.10 6802
CauchoStatus yes
```

Virtual Sites (Virtual Hosts)

If IIS is managing multiple virtual sites (everyone else calls them virtual hosts), then you need to configure IIS to use the isapi_srun.dll filter for each virtual site. Configure IIS to have the virtual directory `/scripts` for each virtual site point to `d:\inetpub\scripts` , so that each virtual site uses the isapiu_srun.dll.

Resin is configured to recognize virtual hosts with the `id` attribute of .

```
<resin xmlns="http://caucho.com/ns/resin">
  ...
  <server>
    ...

    <host id="foo.com">
      ...
    </host>

    <host id="bar.com">
      ...
    </host>

    <host id="baz.com">
      ...
    </host>

  </server>
</resin>
```

resin.xml with virtual hosts

Resin recognizes which host to use by examing the url. With the above example, a url of `http://foo.com/some/path` will use host `id="foo.com"` and a url of `http://bar.com/some/path` will use host `id="bar.com"` .

Virtual Sites with different JVM's

If a seperate JVM for each virtual site is desired, a seperate `resin.ini` is used for each virtual site. The resin.ini file is placed in the `scripts` directory.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

  <server port="6800"/>

  ...
  <host id="*">
    ...
  </host>

</cluster>
</resin>
```

resin-foo.xml

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

  <server port="6801"/>

  ...
  <host id="*">
    ...
  </host>
</cluster>
</resin>
```

resin-bar.xml

```
win> resin-3.2.x/resin.exe -install-as "Resin-foo" \
    -conf resin-foo.xml -Xrs
win> resin-3.2.x/resin.exe -install-as "Resin-bar" \
    -conf resin-bar.xml -Xrs
```

installing a servlet runner for each virtual site

```
ResinConfigServer localhost 6802
```

resin.ini for IIS virtual site foo.com

```
ResinConfigServer localhost 6803
```

resin.ini for IIS virtual site bar.com

The ResinConfigServer tells the isapi_srun.dll the port number to use to connect to the Resin instance. You can change the host from `localhost` to a backend server. You can also add multiple `ResinConfigServer` items to cluster the configuration.

resin-foo.xml and resin-bar.xml contain a <host id="*">, you do not need to specify the host name because each conf/JVM is only going to receive requests from a particular virtual site (because of the unique resin.ini files).

Troubleshooting

1. Check your configuration with the standalone web server. In other words, add a <http port='8080'/> block and browse http://localhost:8080.
2. Check `http://localhost/caucho-status` . That will tell if the ISAPI filter/extension is properly installed.
3. Each server should be green and the mappings should match your resin.xml.

4. If caucho-status fails entirely, the problem is in the `isapi_srun` installation. Try `http://localhost/scripts/isapi_srun.dll/caucho-status` directly (bypassing the filter). If this fails, IIS can't find isapi_srun.dll.
   - Check that isapi_srun.dll is in `c:\inetpub\scripts.`
   - Make sure that both IIS and the underlying NTFS file system have permissions set appropriately for isapi_srun.dll.
   - Make sure that your IIS host has a mapping from `/scripts` to `c:\inetpub\scripts` and that the `/scripts` has execute permissions.
   - IIS 6 users may need to take additional steps.
5. If you've created a new IIS web site, you need to create a virtual directory `/scripts` pointing to the `d:\inetpub\scripts` directory.
6. If caucho-status shows the wrong mappings, there's something wrong with the resin.xml.
7. If caucho-status shows a red servlet runner, then resin.exe hasn't properly started.
8. If you get a "cannot connect to servlet engine", caucho-status will show red, and resin.exe hasn't started properly.
9. If resin.exe doesn't start properly, you should look at the logs in resin3.2/log. You should start `resin.exe -verbose` to get more information.
10. If you get Resin's file not found, the IIS configuration is good but the resin.xml probably points to the wrong directories.

Troubleshooting IIS 6

IIS 6/Windows 2003 users may need to perform additional steps.

- Make sure that the System account has suffiicient privleges to read the `C:\InetPub` and `C:\InetPub\Scripts` directory and the `isapi_srun.dll` .
- Check the 'Web Service Extensions' listed in the 'Internet Service Manager' to make sure that Resin is listed as a Web Service Extension and has a status of "enabled". You may need to click "add a new web service extension...", under Extension name add .jsp or whatever your file extension is, click Add and browse to the isapi_srun.dll, check the "Set extension status to allowed box", click OK.
- Check that the user specified as the "application pool identity" for Resin has read/write permission to the Resin installation directory. In the Internet Service Manager, open the Properties dialog for "Application Pools".

Find the User on the "Identity" tab, it may be the user named "Network Service" in the drop-down list associated with the radio button labeled "predefined". Then check physical file permissions on the Resin installation directory and all its subdirectories and subfiles, to ensure that that user has read/write permission status is "Enabled".

2.6 How the Plugins Dispatch to Resin

The web server plugins (mod_caucho and isapi_srun) have two main tasks:

1. Select urls to dispatch to the Java process
2. Pass the request and retrieve the response from the Java process.

Note: "mod_caucho" is used to mean all the plugins. All of the plugins work the same, so "mod_caucho" is just a shorthand for "mod_caucho and isapi_srun".

ResinConfigServer

mod_caucho discovers its configuration by contacting the ResinConfigServer specified in the httpd.conf or resin.ini. The ResinConfigServer can be any Resin server. When a user requests a URL, mod_caucho uses the configuration it has determined from the ResinConfigServer to determine whether Resin or Apache should handle the request. That decision is based on the configuration in the ResinConfigServer's resin.xml.

servlet-mapping selects URLs

The servlet-mapping tag selects the URLs to send to Resin. <host> and <web-app> group the servlet-mapping tags.

url-pattern

servlet-mapping's url-pattern selects the URLs to pass to Resin. servlet-mapping and url-pattern are part of the Servlet 2.3 standard, so there are many references explaining how it works.

url-pattern can take one of four forms:

- " `/` " matches all URLs. Use this to pass all requests to Resin.
- " `/prefix/url/*` " matches any URL starting with `/prefix/url` , including `prefix/url` itself. It does not match `/prefix/urlfoo` because any slash must immediately follow `url`
- " `/exact/path` " matches only the exact path. In other words, it will not match `/exact/path/bogus` .
- " `*.ext` " matches any URL with the extension `ext` . Resin allows path-infos, so `/foo/bar.ext/path/info` will also match.

url-regexp

Note: mod_caucho does not understand regular expressions. If you put regular expressions in your resin.xml, mod_caucho will not send the request to Resin. Apache will handle the request itself.

If you want to use regular expressions in servlet-mapping, web-app, or hosts, you must use Apache-specific configuration to send the request to Resin. You can see this by looking at /caucho-status. /caucho-status will not display any regular expressions.

special servlet-mappings

There are two special servlet-names which only affect the plugins: `plugin_match` and `plugin_ignore` .

`plugin_match` will direct a request to Resin. The servlet engine itself will ignore the plugin_match directive. You can use plugin_match to direct an entire subtree to Resin, e.g. to workaround the regexp limitation, but allow Resin's other servlet-mapping directives to control which servlets are used.

`plugin_ignore` keeps the request at on the web server. So you could create a directory `/static` where all documents, including JSPs are served by the web server.

```
<!-- send everything under /resin to Resin -->
<servlet-mapping url-pattern='/resin/*'
                 servlet-name='plugin_match'/>

<!-- keep everything under /static at the web server -->
<servlet-mapping url-pattern='/static/*'
                 servlet-name='plugin_ignore'/>
```

<web-app>

web-apps collect servlets and JSP files into separate applications. All the servlet-mappings in a web-app apply only to the URL suffix.

In the following example, every URL starting with /prefix/url maps to the web-app. The servlet-mapping only applies to URLs matching the prefix.

```
...
<web-app id='/prefix/url'>
  <servlet-mapping url-pattern='*.foo' .../>
</web-app>
..
```

In the exaple, mod_caucho will match any URL matching /prefix/url/*.foo. /prefix/url/bar.foo will match, but /test/bar.foo will not match.**Note:** Resin standalone allows a `regexp` attribute instead of an id. Because mod_caucho

does not understand regexps, it will ignore any web-app with a `regexp` attribute.**Note:** web.xml files and war files are treated exactly the same as web-apps in the resin.xml.

<host>

host blocks configure virtual hosts. There's a bit of extra work for virtual hosts that we'll ignore here. (Basically, you need to add Apache `ServerName` directives so Resin knows the name of the virtual host.)

For dispatching, a host block gathers a set of web-apps. Each host will match a different set of URLs, depending on the web-app configuration. The default host matches any host not matched by a specific rule.

As usual, /caucho-status will show the URLs matched for each host.**Note:** mod_caucho does not understand the host `regexp` attribute. It will ignore all hosts using `regexp` . To get around this, you can either configure Apache directly (see below), or configure the default host with the same set of servlet-mappings. Since mod_caucho will use the default host if no others match, it will send the right requests to Resin.

/caucho-status shows mod_caucho's URLs

The special URL `/caucho-status` is invaluable in debugging Resin configurations. `/caucho-status` displays all the resin.xml patterns, so you can easily scan it to see which URLs mod_caucho is sending to Resin and which ones are handled by Apache.

Dispatching using Apache's http.conf

You can configure Apache directly, instead of letting mod_caucho dispatch from the resin.xml file. If you use this method, you need to make sure you match the Apache configuration with the Resin configuration.**Note:** This technique uses Apache-specific features, so it's not directly applicable to IIS or iPlanet.

Apache's `Location` and `SetHandler` directives send requests to Resin. The mod_caucho handler is `caucho-request` .

```
LoadModule caucho_module libexec/mod_caucho.so
AddModule mod_caucho.c

CauchoHost localhost 6802
AddHandler caucho-request jsp
<Location /servlet/*>
   SetHandler caucho-request
</Location>
```

httpd.conf

Because Apache's `SetHandler` is external to mod_caucho, /caucho-status will not show any `SetHandler` dispatching.

Chapter 3

Command-Line

3.1 Command-Line Configuration

./configure options

The `./configure; make; make install` step is important for all Unix users. It configures and compiles low level JNI code that enables Resin to provide a number of features not normally available to Java programs and also provides significant performance improvements.

The most commonly used options for `./configure` are documented below, the full set of available command line options is available by running `./configure --help` .

–help	Help for all ./configure command line options
–enable-64bit	Compiles the JNI using 64-bits, requires <jvm-arg>-d64</jvm-arg> entry in resin.xml
–enable-ssl	Enable OpenSSL, see the OpenSSL documentation for details.
–with-apxs= /path/to/apxs	Enable Apache integration and produce mod_caucho

The 64-bit JNI compilation must match the JDK you're using, i.e. you'll need to add a <jvm-arg>-d64</jvm-arg> entry in resin.xml to indicate that the jvm should start in 64-bit mode.

Startup Options

As of Resin 3.1, startup options should be declared in the configuration file. However, some startup options are available via the command line.

Command-line arguments

ARGUMENT	MEANING	DEFAULT
-conf `xxx`	Selects the Resin configuration file	conf/resin.xml
-server `xxx`	Selects the <server> in the resin.xml	""
-verbose	Show the Java environment before starting Resin.	off
start	Starts Resin as a daemon, starting the watchdog if necessary	n/a
status	Show the status of Resin as a daemon.	n/a
stop	Stops Resin as a daemon by contacting the watchdog.	n/a
restart	Restarts Resin as a daemon by contacting the watchdog.	n/a
kill	Kill Resin as a daemon by contacting the watchdog, a killed process is destroyed and not allowed to clean up or finish pending connections.	n/a
shutdown	Shutdown the watchdog and all of the Resin daemons.	n/a
-install	(Windows) install Resin as a service (but doesn't automatically start.)	n/a
-install-as `xxx`	(Windows) install Resin as a named service (but doesn't automatically start.)	n/a

-remove	(Windows) install Resin as a service (but doesn't automatically start.)	n/a
-remove-as `xxx`	(Windows) remove Resin as a named service (but doesn't automatically start.)	n/a
-resin-home `xxx`	Deprecated. Sets the Resin home directory. Use environment variable `RESIN_HOME` or `<jvm-arg>-Dresin.hom` in resin.xml.	The parent directory of resin.jar

JDK arguments

Resin 3.1 has moved all JDK arguments into the resin.xml file, in the <jvm-arg> tag. Because the Resin 3.1 watchdog starts each Resin server instance, it can pass the arguments defined in the configuration file to the JVM. By moving the Java arguments to the configuration file, server configuration is easier and more maintainable.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">

    <server-default>
      <jvm-arg>-Xms32m</jvm-arg>
      <jvm-arg>-Xmx512m</jvm-arg>
      <jvm-arg>-Xss1m</jvm-arg>
      <jvm-arg>-verbosegc</jvm-arg>
      <jvm-arg>-Dfoo=bar</jvm-arg>
      <jvm-arg>-agentlib:resin</jvm-arg>
      <jvm-arg>-Xdebug</jvm-arg>

      <http port="8080"/>
    </server-default>

    <server id="a" address="192.168.2.1" port="6800"/>

    ...
  </cluster>
</resin>
```

resin.xml with Java arguments

Chapter 4

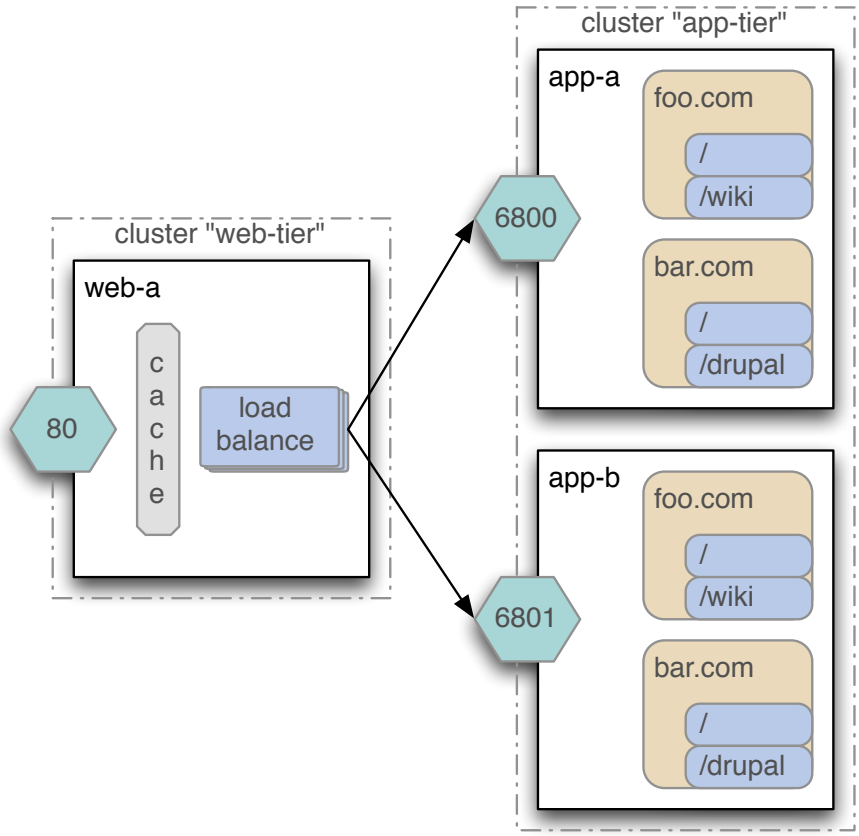
Admin Guide

4.1 User Guide: Administration

Terminology

- **cluster** - A set of *servers* configured to serve identical content. Even a solo server will belong to its own cluster.
- **default host** - A virtual host configured to handle any requests not maching a specified host.
- **host** - An internet domain like www.slytherin.com whose content is handled by a cluster. Also called virtual host.
- **keepalive** - A HTTP connection kept open after a request finishes so the next request to the server will be more efficient.
- **load-balancer** - A Resin *server* which forwards requests to an application cluster for increased scalability and reliability.
- **machine** - A physical computer. Multiple *servers* can run on the same machine.
- **port** - A server's internet connection handling a specific protocol, e.g. a HTTP port 80 or HTTPS port 443 or XMPP (Jabber) port 5222.
- **proxy cache** - A content cache in front of a web application, storing the content and returning it quickly without running the application. Speedups for some applications can be 10x or 100x.
- **server** - A Resin instance running on a single JVM. One or more servers can run on the same *machine*.
- **servlet** - A Java program responsible for serving web content. All content is handled by a servlet, including static files, JSPs and PHP content.

- **thread** - An active execution of a Java program. Because Java is multi-threaded, multiple Java programs are running simultaneously. For example, each HTTP user/request is handled by its own Java thread.
- **watchdog** - A Resin Java process responsible for starting and restarting Resin *servers* for reliability.
- **webapp** - A web application is a content collection, like a Drupal or Mediawiki site. All content belongs to a webapp.



In the previous figure:

- The *servers* are "web-a", "app-a" and "app-b".
- The *clusters* are "web-tier" and "app-tier".
- The *virtual hosts* are "foo.com" and "bar.com".
- The *webapps* are "/", "/wiki" and "/drupal".

- Both *servers* "app-a" and "app-b" in the *cluster* "app-tier" serve identical content, i.e. the same *virtual hosts* and *webapps*.
- web-a has a *proxy-cache* and *load balancer*.
- web-a listens to HTTP *port* 80.
- app-a and app-b listen to cluster *port* 6800 and 6801.
- All three *servers* could be on the same *machine* or on separate *machines*.

Dispatching Content

All HTTP content in Resin is ultimately handled by a servlet. If Resin can't find a servlet for a URL, it will return a `404 Not Found` to the browser. So, if you see an unexpected `404 Not Found` , you not only need to check that the file exists, but make sure that the servlet and its URL mapping is properly configured. If you turn on logging to "finer", you can trace the request to figure out why the servlet is not getting called.

To match up the URL to its final servlet and the content, you need all of the following properly configured:

1. **Server**. The Resin server must be active to do anything. Because servers belong to a cluster, you'll automatically have a cluster even if it only has a single server.
2. **HTTP port**. The server must be listening to the internet for HTTP requests just to get started. If the HTTP port is missing or misconfigured, you will get connection failure messages because the browser cannot connect to your server at all.
3. **Host**. Resin must first match the <host> specified by the HTTP request, e.g. www.slytherin.com. If no hosts match, Resin will use the default host. If no default host exists, Resin will return a `404 Not Found` to the browser.
4. **WebApp**. Inside the host, Resin finds a web-application to handle the request by looking for the <web-app> with the longest URL prefix. So, `http://www.slytherin.com/drupal/index.php` might match the `/drupal/` . The ROOT web-app matches all URLs. If Resin can't find a web-app, it will return `404 Not Found` to the browser.
5. **servlet-mapping**. Inside the web-app, Resin searches for a <servlet-mapping> matching the URL. For example, `test.php` would match the `QuercusServlet` and `test.jsp` would match the JSP servlet. If none match, Resin will try the default mapping, which is normally the `FileServlet` to handle static pages. If the default servlet isn't configured, Resin will return a `404 Not Found` to the browser.

6. **Servlet**. Finally, the selected servlet processes the request and returns the content. The servlet itself might not find the requested content, e.g. if `/foo.php` does not exist in the expected location. The servlet itself is responsible for error handling, but most servlets will return a `404 Not Found` if any expected files are missing.

Based on Resin's dispatching flow, here's a minimal resin.xml to serve some content out of `/var/www/htdocs` . Resin's philosophy of configuration files is that 1) for maintainability, all configuration should be traceable to the resin.xml, i.e. no magic defaults or hidden state are allowed and 2) for security, if something is not configured, it doesn't exist. In Resin, you need to enable things explicitly, not disable hidden defaults. The slight extra verbosity is outweighed by the improved security and maintainability.

The following resin.xml specifies an Apache-style structure where all content is served from the `/var/www/htdocs` directory, and is useful when upgrading from an old PHP site to use Quercus for security and performance. When organizing a site from scratch, you'll typically use a more structured dynamic hosting directory structure.

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">

  <cluster id="app-tier">
    <development-mode-error-page/>

    <server id="" address="127.0.0.1" port="6800">
      <http port="8080"/>
    </server>

    <resin:import path="/etc/resin/app-default.xml"/>

    <host id="">
      <web-app id="">
        <root-directory>/var/www/htdocs</root-directory>
      </web-app>
    </host>
  </cluster>

</resin>
```

Example: minimal /etc/resin/resin.xml for HTTP

- <resin> starts a Resin configuration file and declares the validation namespaces.
- <cluster> encloses the single-server cluster containing our content.
- <development-mode-error-page> reports configuration and runtime errors to the browser, which is very helpful during development. On a production server, you may want to remove this tag so errors don't expose information to the internet.

- <server> configures the Resin server, including its ports. The `id` matches the command-line `-server` argument at startup. The `address` and `port` open Resin's cluster port, which is used for deployment, management, clustering, and distributed caching.

- <http> listens for HTTP requests. Production servers will change the port to 80.

- <resin:import> defines the standard servlet like JSP, PHP and the static file servlet. If you omit this <resin:import>, Resin will return `404 Not Found` because the <servlet-mapping> and servlets would not be defined.

- <host> defines a default virtual host. The default host will handle any host domain given by a HTTP request.

- <web-app> defines a ROOT web-app, serving all URLs for the host. The servlets in the web-app are defined by the `app-default.xml` specified by the <resin:import>.

- <root-directory> specifies the content directory, here matching a standard Apache directory.

rewrite-dispatch replaces mod_rewrite

For many applications like Drupal and Mediawiki, it's important to rewrite a user-friendly URL to an internal servlet or PHP URL. Resin's <rewrite-dispatch> replaces the capabilities of mod_rewrite for Apache.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <rewrite-dispatch>
    <dispatch regexp="\.(php|gif|css|jpg|png|ico|js|htm|html)"/>
    <forward regexp="^/" target="/index.php?q="/>
  </rewrite-dispatch>

</web-app>
```

Example: WEB-INF/resin-web.xml Drupal rewriting

- <web-app> configures the `/drupal` web-application.

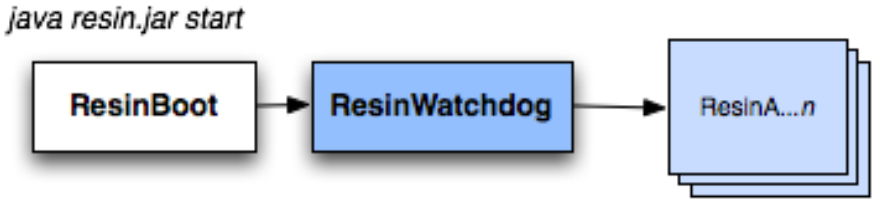
- <rewrite-dispatch> starts the URL rewriting configuration.

- <dispatch> passes the URL untouched for content handling, e.g. static files or php.

- <forward> rewrites the URL and forwards.

Resin Processes

- **command-line** - the command-line process (`ResinBoot`) only exists long enough to send a message to the *watchdog* or the Resin process, like a "start", "deploy", or "stop". If necessary, the command-line process will start the watchdog.
- **watchdog** - the watchdog process manages and monitors *Resin server* processes. If the Resin server exits, the watchdog will automatically restart Resin, providing extra reliability in case of server failure.
- **Resin server** - the Resin server handles the HTTP requests and serves the content.



Because the Resin server is started as a child of the watchdog process, its own JVM arguments come from the resin.xml, not from the command-line of `ResinBoot` . The JVM configuration arguments are supplied as <jvm-arg> in the <server> configuration.

Files and Directories

Resin's configuration allows for great flexibility in choosing a directory structure for your site, or adapting Resin to your existing site configuration if you're upgrading from Apache/PHP for a Drupal or MediaWiki site. There are three main directory styles:

- **Resin dynamic virtual host** - the most general solution, letting you easily add virtual hosts without modifying the configuration files.
- **Apache upgrade** - all content is placed in a `/var/www/htdocs` directory.
- **Servlet war style** - all content is deployed with .war files in a webapps directory. The .war file is automatically expanded and deployed from the webapps.

Dynamic Virtual Host

In the dynamic virtual host configuration, your content is placed in `/var/www/hosts/foo.com/webapps/ROOT` . `/var/www` is the standard location for HTTP documents. `hosts/foo.com` contains your virtual host root directory. `webapps/ROOT` contains the root web-app for the virtual host.

Adding a new virtual host is easy just by adding a new `hosts/bar.com` directory. The special directory `hosts/default` creates a default virtual host, which will serve pages for anything not matching a specific host. If you need to configure the host, or add <host-alias> tags, you can add them to a `hosts/bar.com/host.xml` file.

Deploying a web-application is also straightforward by adding a .war file to `webapps/foo.war` or `webapps/ROOT.war` . `webapps/ROOT` is a special web-app which handles the root URL, "/".

In a standard Unix deployment, your configuration files go in `/etc/resin` and typically includes the `resin.xml` and the `app-default.xml` . `app-default.xml` contains standard configuration like the static file servlet, JSP, PHP servlets, as well as the standard mime-type mappings.

The Resin jars, JNI libraries and licenses go in `/usr/lib/resin` , typically owned by root. The `resin` directory itself will normally be a symbolic link to the Resin version, so you can upgrade easily.

```
/etc/resin/
  resin.xml                - main Resin configuration file
  app-default.xml          - default servlet configuration
  admin-users.xml          - /resin-admin users

/etc/init.d/resin          - startup script

/usr/lib/resin/            - ${resin.home}
   lib/*.jar               - Resin's Java classes
   libexec/*.so            - native JNI libraries
   libexec64/*.so          - JNI for 64-bit systems
   licenses/*              - Resin Pro licenses
   win32/*.dll             - Win32 JNI
   win64/*.dll             - Win64 JNI

/var/www/                  - ${resin.root}
   admin/                  - Resin database and cache storage
   doc/admin/              - /resin-admin source
   ivy2/                   - ivy2 jar repository
   lib/                    - Shared application jars
   log/                    - watchdog and Resin debug logging
   hosts/foo.com/          - dynamic host contents
     log/                  - access logging per virtual host
     host.xml              - optional host-specific configuration
     webapps/ROOT/         - web-app content
       index.php           - web-app content
       WEB-INF/            - web-app classes and configuration
         web.xml           - servlet-standard configuration
         resin-web.xml     - Resin-specific configuration
         lib/*.jar         - application jars
         classes/**/*.class - application classes
         work/             - work directory for JSP, PHP, etc.
```

Example: Unix dynamic virtual host

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">
  <root-directory>/var/www</root-directory>

  <cluster id="">
    <server id="" address="127.0.0.1" port="6800">
      <http port="8080"/>
    </server>

    <resin:import path="/etc/resin/app-default.xml"/>

    <host-default>
      <resin:import path="host.xml" optional="true"/>

      <web-app-deploy path="webapps"/>
    </host-default>

    <host-deploy path="hosts"/>
  </cluster>
</resin>
```

Example: /etc/resin/resin.xml

Apache Upgrade

In an Apache upgrade, you have an existing site that you're upgrading to use Quercus or just evaluating the performance improvement for an existing side. The content is placed in `/var/www/htdocs` directory, and there's only a single virtual host and single root web-app.

For Apache-style sites with multiple virtual hosts, you can add new <host> tags for each host, and add a <root-directory> to set the host's root.

```
/etc/resin/
  resin.xml                 - main Resin configuration file
  app-default.xml           - default servlet configuration

/etc/init.d/resin           - startup script

/usr/local/share/resin/     - ${resin.home}
   lib/*.jar                - Resin's Java classes
   ...

/var/www/                   - ${resin.root}
   admin/                   - Resin database and cache storage
   log/                     - watchdog and Resin debug logging
   logs/                    - HTTP access logging
   htdocs/
     index.php              - web-app content
     WEB-INF/               - web-app classes and configuration
       resin-web.xml        - Resin-specific configuration
       lib/*.jar            - application jars
       classes/**/*.class   - application classes
       work/                - work directory for JSP, PHP, etc.
```

Example: Unix Apache upgrade

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">
  <root-directory>/var/www</root-directory>

  <cluster id="">
    <server id="" address="127.0.0.1" port="6800">
      <http port="8080"/>
    </server>

    <resin:import path="/etc/resin/app-default.xml"/>

    <host id="">

      <web-app id="" root-directory="htdocs"/>

    </host>
  </cluster>
</resin>
```

Example: /etc/resin/resin.xml

Java App-Server directory structure

A common Java app-server deployment structure looks like a mixture of the full dynamic host configuration and the Apache-style static configuration. The virtual hosts are configured statically like Apache, but webapps are created dynamically by dropping .war files in a `webapps` directory.

```
/etc/resin/
  resin.xml                 - main Resin configuration file
  app-default.xml           - default servlet configuration

/etc/init.d/resin           - startup script

/usr/local/share/resin/     - ${resin.home}
   lib/*.jar                - Resin's Java classes
   ...

/var/www/                   - ${resin.root}
   admin/                   - Resin database and cache storage
   log/                     - watchdog and Resin debug logging
   logs/                    - HTTP access logging
   webapps/
     ROOT/                  - web-app for "/"
       index.php            - web-app content
       WEB-INF/             - web-app classes and configuration
         resin-web.xml      - Resin-specific configuration
         lib/*.jar          - application jars
         classes/**/*.class - application classes
         work/              - work directory for JSP, PHP, etc.
     wiki/                  - web-app for "/wiki"
       ...
     drupal/                - web-app for "/wiki"
       index.php
```

Example: Java app-server structure

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">
  <root-directory>/var/www</root-directory>

  <cluster id="">
    <server id="" address="127.0.0.1" port="6800">
      <http port="8080"/>
    </server>

    <resin:import path="/etc/resin/app-default.xml"/>

    <host id="">

      <web-app-deploy path="webapps"/>

    </host>
  </cluster>
</resin>
```

Example: /etc/resin/resin.xml

Threads, Memory and Sockets

A Resin server needs to have enough memory, threads, and sockets to handle the user load. For the most part, the defaults provide a good starting point, but it's a good idea to become familiar with the primary resources.

- **memory** - divided into heap, thread stack, PermGen space, and some specialized JVM/JNI space. With a 32-bit machine, you are likely to run into virtual memory limits. Even with a 64-bit machine, you need enough swap to handle your virtual memory size.

- **sockets** - a TCP connection to the browser. Each user uses one or sockets as long as the browser is connected. Sockets use up **file descriptors** on Unix, which are also used by database connections, and open files. The *netstat* command is important for threading.

- **threads** - an program execution, taking up CPU and virtual memory (for the thread stack). Active requests use CPU, while even blocked threads take up memory.

Memory

Memory is the most important configuration item for the machine. You need enough virtual memory to handle heap, stack, and permanent memory, and enough physical memory to avoid swapping. If the JVM runs out of memory, it may throw `OutOfMemoryError` and force a restart of the server.

- **heap** - Java memory used for all application objects. Heap memory is garbage collected periodically, causing the JVM to stop or slow while finding unused memory. The maximum heap is set by the `-Xmx512m` command-line argument in <jvm-arg>.

- **PermGen (permanent generation)** is JVM memory for permanent data like Java class code, JNI code, shared libraries, mmapped memory (jars), and the JVM itself. Configured by `-XX:PermGen=128M` . The JVM default usually needs to be increased.

- **thread stack** - Each thread takes up virtual memory for its execution stack. Since the default is 1M and having 1000 threads is fairly common, you can use 1G of virtual memory just for the thread stack. The thread stack is configured by `-Xss1m`

- **virtual memory** is total memory allocated to the JVM by the operating system. VM = heap + stack + PermGen. For a 32-bit system, you might have as little as 2G for all the heap, stack, and PermGen. You also need to have enough **swap** memory to handle your virtual memory, especially if you have a large heap, lots of threads, or even lots of large jars.

Notifications

Resin can notify you by email when it detects something wrong with the server. The main mechanism is adding a new logging handler sending mail for warning and severe messages. When a log entry is added at the WARNING or SEVERE level, Resin will send a mail containing the log information. You can then look at the administration console to debug any problems.

The configuration is described in the logging configuration section.

```
<resin xmlns="http://caucho.com/ns/resin">

  <log-handler name="" level="warning" uri="mail:">
    <init>
      <to>admin@foo.com</to>
      <properties>
        mail.smtp.host=127.0.0.1
        mail.smtp.port=25
      </properties>
    </init>
  </log-handler>

</resin>
```

Example: resin.xml with mail logging

Chapter 5

Watchdog

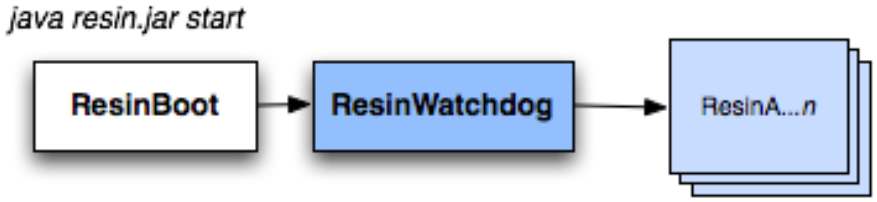
5.1 Resin Watchdog

Overview

Because the watchdog runs quietly as a separate service, most of the time you won't need to pay attention to the watchdog at all. The standard configuration launches on watchdog per machine which monitors all the Resin JVMs on that matchine, so most sites will not need to change any watchdog configuration.

The main management tasks where you might need to pay attention to the watchdog is shutting it down if something is severely wrong with the machine, and checking the watchdog logs for Resin restart events.

The watchdog automatically restarts Resin if the Resin JVM ever crashes or exits. So if you want to stop Resin, you need to tell the watchdog to stop the instance, or you can stop the watchdog entirely. The watchdog is typically controlled by the resin.jar main program (`ResinBoot`), which has commands to start, stop, and restart Resin instances as well as reporting the watchdog status.



While most users will use the watchdog automatically with no extra configuration, ISPs and larger and complicated sites can create a specialised watchdog.xml with a <watchdog-manager> tag to control the watchdog at a much finer level. The <watchdog-manager> lets an ISP run the watchdog under its own control, and specify exactly the command-line parameters for their users'

Resin instances, including the ability to create secure chroot instances for their users. Typically, the watchdog will run as root, while the user instances will run with their respective user ids.

command-line

The watchdog is controlled on the command-line using resin.jar's main class, `ResinBoot` . The major operations are: start, stop, restart, shutdown and status.

start

The "start" command starts a new Resin instance with the given server id. `ResinBoot` will first try to contact the watchdog on the current machine, and start a new watchdog if necessary. The server id must be unique for all servers defined in the resin.xml.

```
resin-3.2.x> java -jar lib/resin.jar -conf conf/test.conf -server a start

Resin/3.2.x started -server 'a' for watchdog at 127.0.0.1:6700
```

Example: watchdog start

stop

The "stop" command stops the Resin instance with the given server id. If the stopped instances is the last one managed by the watchdog, the watchdog will automatically exit. If no `-server` is specified, the watchdog defaults to `-server ""` .

```
resin-3.2.x> java -jar lib/resin.jar stop

Resin/3.2.x started -server '' for watchdog at 127.0.0.1:6600
```

Example: watchdog stop

status

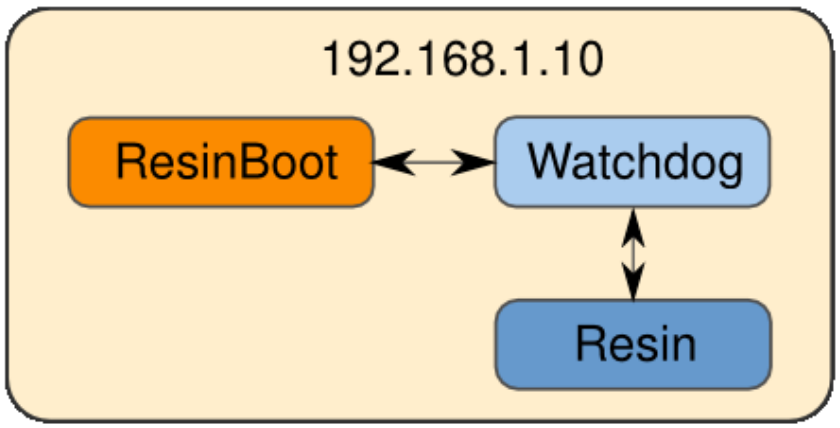
The "status" command summarizes the current Resin instances managed by the watchdog service.

```
resin-3.2.x> java -jar lib/resin.jar status

Resin/3.2.x status for watchdog at 127.0.0.1:6600

server '' : active
  password: missing
  user: ferg
  root: /home/test/resin/
  conf: /etc/resin/resin.xml
```

Example: watchdog status

Single Resin instance

This example shows a single-server site listening to the standard HTTP port 80 and running the server as the "resin" user. In this example, the watchdog typically runs as root so it can bind to the protected port 80, while the Resin instance runs as "resin" for security.



Since this configuration uses the default, the watchdog listens to port 6600 for commands.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

  <server id="app-a" address="127.0.0.1">
    <user-name>resin</user-name>
    <group-name>resin</group-name>

    <http port="80"/>
  </server>

  <resin:import path="${resin.home}/conf/app-default.xml"/>

  <host id="">
    <web-app id="" path="/var/www/htdocs"/>
  </host>

</cluster>
</resin>
```

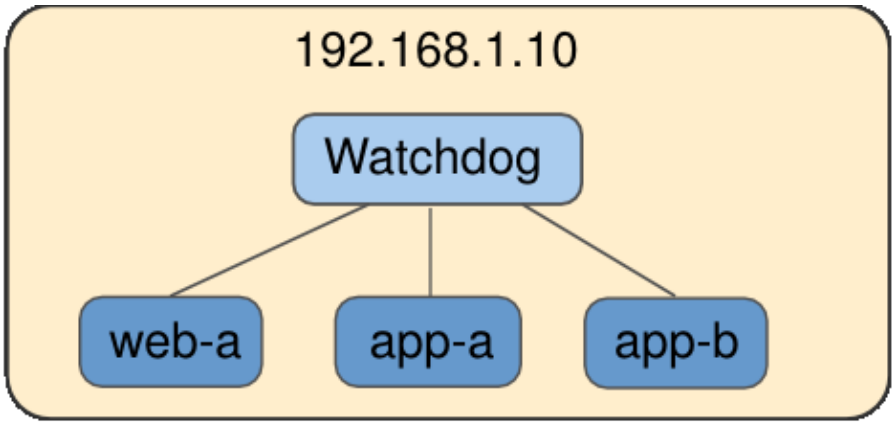
Example: /etc/resin/resin.xml

Single machine load balance with shared watchdog

When running multiple instances of Resin on the same machine, one watchdog-manager typically handles all the instances. The server id will select which instance to start or stop.

In this example, there is one web-tier server acting as a load-balancer and two app-tier servers handling the backend data, all on a single machine. A site might want multiple app-tier servers for more reliable maintenance and upgrades. While one server is down, traffic can be handled by a second server.

The example uses default watchdog configuration from the standard resin.xml file. The watchdog process and `ResinBoot` will both read the resin.xml file for the server configuration, so there's no explicit watchdog configuration necessary. The watchdog detects that multiple servers are running on the same machine and manages all of them automatically.


192.168.1.10
Watchdog
web-a
app-a
app-b

```
<resin xmlns="http://caucho.com/ns/resin">

<cluster id="app-tier">

  <server-default>
    <user-name>resin</user-name>
    <group-name>resin</group-name>
  </server-default>

  <server id="app-a" address="192.168.1.10" port="6810"/>
  <server id="app-b" address="192.168.1.10" port="6811"/>

  <host id="">
    <web-app id="" path="/var/www/htdocs"/>
  </host>

</cluster>

<cluster id="web-tier">

  <server-default>
    <user-name>resin</user-name>
    <group-name>resin</group-name>
  </server-default>

  <server id="web-a" address="192.168.1.10" port="6800">
    <http port="80"/>
  </server>

  <host id="">
    <web-app id="">
      <rewrite-redirect>
        <load-balance regexp="" cluster="app-tier"/>
      </rewrite-redirect>
    </web-app>
  </host>

</cluster>

</resin>
```

Example: /etc/resin/resin.xml

Single machine load balance with distinct watchdog

In some cases, it's best to let each Resin instance have its own watchdog, for example when multiple users are sharing the same machine. Each <server> block configures a separate <watchdog-port>. Because the watchdog will read the resin.xml and use the <server> block matching the `-server id` command-line argument, each watchdog will start with it's own port.

```
<resin xmlns="http://caucho.com/ns/resin">

<cluster id="app-tier">

  <server-default>
    <user-name>resin</user-name>
    <group-name>resin</group-name>
  </server-default>

  <server id="app-a" address="192.168.1.10" port="6810">
    <watchdog-port>6700</watchdog-port>

    <http port="8080"/>
  </server>

  <server id="app-b" address="192.168.1.10" port="6811">
    <watchdog-port>6701</watchdog-port>

    <http port="8081"/>
  </server>

  <host id="">
    <web-app id="" path="/var/www/htdocs"/>
  </host>

</cluster>

</resin>
```

Example: /etc/resin/resin.xml

In the previous example, starting Resin with `-server app-a` will start a watchdog at port 6700. Starting Resin with `-server app-b` will start the watchdog at port 6701.

```
resin-3.2.x> java -jar lib/resin.jar -server app-b start
```

Example: starting app-b with watchdog-port=6701

ISP watchdog management

In a situation like an ISP, you may wish to have a separate configuration file for the watchdog, which launches Resin instances for different users. In this case, you will want to make sure the watchdog.xml is not readable by the users, and make sure to set a management user (see resin-security).

- Start and restart the user's Resin JVM
- Set JVM parameters and Java executable
- Set the Resin instance root-directory

- setuid user-name and group-name
- Set the resin.xml configuration (must be readable by the user)
- Open protected ports like port 80
- Optional chroot for additional security

The watchdog will launch the Resin instance with the given user as a setuid. It will also open any necessary protected ports, e.g. port 80.

```
<resin xmlns="http://caucho.com/ns/resin">

<management>
  <user name="harry" password="MD5HASH=="/>
</management>

<watchdog-manager>

  <watchdog-default>
    <jvm-arg>-Xmx256m</jvm-arg>
  </watchdog-default>

  <watchdog id="user_1">
    <user-name>user_1</user-name>
    <group-name>group_1</group-name>

    <resin-xml>/home/user_1/conf/resin.xml</resin-conf>
    <resin-root>/home/user_1/www</resin-root>

    <open-port address="192.168.1.10" port="80"/>
  </watchdog>

  ...

  <watchdog id="user_n">
    <user-name>user_n</user-name>
    <group-name>group_n</group-name>

    <resin-conf>/home/user_n/conf/resin.xml</resin-conf>
    <resin-root>/home/user_n/www</resin-root>

    <open-port address="192.168.1.240" port="80"/>
  </watchdog>

</watchdog-manager>

</resin>
```

Example: /etc/resin/watchdog.xml

Management/JMX

The watchdog publishes the watchdog instances to JMX with the JMX name "resin:type=Watchdog,name=a". With a JMX monitoring tool like jconsole,

you can view and manage the watchdog instances.

Chapter 6

Virtual Hosts

6.1 Virtual Hosting

Overview

Virtual hosts are multiple internet domains served by the same Resin server. Because one JVM handles all the domains, its more memory and processing efficient, as well as sharing IP addresses. With Resin, adding virtual hosts can as easy as creating a directory like `/var/www/hosts/foo.com` and setting up the DNS name. Explicit virtual host is also possible to match existing layouts, like matching a `/var/www/htdocs` configuration when migrating a PHP mediawiki or wordpress site to use Quercus for security and performance.

The virtual host will contain one or more web-apps to serve the host's contents. Simple sites will use a fixed root webapp, like the Apache-style `/var/www/htdocs` . More complicated sites can use a `webapps` -style directory.

Each virtual host belongs to a Resin <cluster>, even if the cluster has only a single server.

For example, a Resin server might manage both the `www.gryffindor.com` and `www.slytherin.com` domains, storing the content in separate directories (/var/www/gryffindor and /var/www/slytherin), and using a single IP address for both domains. In this scenario, both www.gryffindor.com and www.slytherin.com are registered with the standard domain name service registry as having the IP address `192.168.0.13` . When a user types in the url `http://www.gryffindor.com/hello.jsp` in their browser, the browser will send the HTTP request to the IP address `192.168.0.13` and send an additional HTTP header for the gryffindor host, "Host: www.gryffindor.com". When Resin receives the request it will grab the host header, and dispatch the request to the configured virtual host.

```
C: GET /test.jsp HTTP/1.1
C: Host: www.gryffindor.com
C:
```

Example: HTTP request headers

1. host name
2. host aliases
3. optional host.xml
4. root directory
5. web-applications
6. configuration environment
7. logging

Dynamic virtual hosts

Resin can deploy virtual hosts automatically by scanning a host deployment directory for virtual host content. Each sub-directory in the `hosts` directory will cause Resin to create a new virtual host. To customize the configuration, you can add a `host.xml` in the host's root directory for shared databases, beans or security, or to add <host-alias> names.

You can add hosts dynamically to a running server just by creating a new host directory. Resin periodically scans the `hosts` directory looking for directory changes. When it detects a new host directory, it will automatically start serving files from the new virtual hosts.

If you add a `default` directory in the `hosts` , Resin will use it to serve all unknown virtual hosts. The default host is handy for simple servers with only a single virtual host and for sites where the virtual host is handled in software, like Drupal. If the `default` directory is missing, Resin will return `404 Not Found` for any unknown virtual hosts.

```
/var/www/hosts/www.gryffindor.com/
                                host.xml
                                log/access.log
                                webapps/ROOT/index.jsp
                                webapps/ROOT/WEB-INF/resin-web.xml

/var/www/hosts/www.slytherin.com/
                               host.xml
                               log/access.log
                               webapps/ROOT/index.php
                               webapps/ROOT/WEB-INF/resin-web.xml

/var/www/hosts/default/
                      host.xml
                      log/access.log
                      webapps/ROOT/index.php
                      webapps/ROOT/WEB-INF/resin-web.xml
```

Example: virtual host directory structure

host-aliasing for dynamic hosts

Often, the same virtual host will respond to multiple names, like `www.slytherin.com` and `slytherin.com` . One name is the primary name and the others are aliases. In Resin, the primary name is configured by the <host-name> tag and aliases are configured by <host-alias>. In a dynamic host configuration, the directory name is used as the `host-name` by default, and aliases are declared in the `host.xml` .

```
<host xmlns="http://caucho.com/ns/resin">
  <host-name>www.slytherin.com</host-name>
  <host-alias>slytherin.com</host-alias>
  <host-alias>quidditch.slytherin.com</host-alias>
</host>
```

Example: www.slytherin.com/host.xml

Since the `host.xml` is shared for all web-applications in the host, you can also use it to configure shared resources like security logins, shared databases, and shared resources.

host-deploy configuration

The <host-deploy> tag configures the dynamic virtual hosting specifying the directory where Resin should scan for virtual hosts. Because Resin does not automatically add default configuration, you will need to also add configuration for the `host.xml` , `app-default.xml` and `web-app-deploy` . Although it's a bit more verbose, the no-default rule makes Resin more secure and debuggable.

If an item like a <web-app> is missing, Resin will return `404 Not Found` for security. Because all configuration is explicit, it's ultimately traceable to the `resin.xml` which makes debugging more reliable.

Shared host configuration goes in the <host-default> tag. In this case, we've added an optional `host.xml` for configuration, an access log in `log/access.log` and a standard `webapps` directory. The standard servlets and file handling come from the `app-default.xml` file. If you omit either the app-default.xml or the webapps, you will see `404 Not Found` for any requests.

The example below is a complete, working `resin.xml` listening to HTTP at port 8080. The cluster consists of a single server. It includes a <development-mode-error-page/> to help debugging the configuration. Many sites will omit the error-page to hide configuration details in case an error occurs on a live site.

```
<resin xmlns="http://caucho.com/ns/resin"
         xmlns:resin="http://caucho.com/ns/resin/core">
<cluster id="app-tier">
  <server id="app-a" address="192.168.1.13" port="6800">
    <http port="8080"/>
  </server>

  <development-mode-error-page/>

  <resin:import path="${__FILE__}/../app-default.xml"/>

  <host-default>
    <resin:import path="host.xml" optional="true"/>

    <access-log path="log/access.log"/>

    <web-app-deploy path="webapps"/>
  </host-default>

  \textbf{\ensuremath{<}host-deploy path="hosts"\ensuremath{>}
  \ensuremath{<}/host-deploy\ensuremath{>}}

</cluster>
</resin>
```

Example: /etc/resin/resin.xml host-deploy configuration

Any directory created in `${resin.root}/hosts` will now become a virtual host. You can also place a .jar file in `${resin.root}/hosts` , it is expanded to become a virtual host.

```
${resin.root}/hosts/www.gryffindor.com/
${resin.root}/hosts/www.gryffindor.com/webapps/ROOT/index.jsp
${resin.root}/hosts/www.gryffindor.com/webapps/foo/index.jsp

${resin.root}/hosts/www.slytherin.com.jar
```

Jar libraries and class files that are shared amongst all webapps in the host can be placed in `lib` and `classes` subdirectories of the host:

```
${resin.root}/hosts/www.gryffindor.com/lib/mysql-connector-java-3.1.0-alpha-bin.jar
${resin.root}/hosts/www.gryffindor.com/classes/example/CustomAuthenticator.java
```

More information is available in the configuration documentation for <host-deploy> and <host-default>.

Explicit Virtual Hosting

In a more structured site, you can take complete control of the virtual host configuration and configure each virtual host explicitly. Existing sites wanting to upgrade to Resin or sites with extra security needs may prefer to configure each <host> in the resin.xml. For example, a PHP Drupal site evaluating Quercus to improve performance and security might use the explicit <host> to point to the existing `/var/www/htdocs` directory.

In the explicit configuration, each virtual host has its own host block. At the very least, each host will define the `id` specifying the host name and a root web-app. A is often used to provide a host specific root for logfiles.

As with the dynamic hosting, servlets and web-apps must be configured either in a <host-default> or explicitly. If they are missing, Resin will return a `404 Not Found` for security. The host id="" is the default host and will serve any request that doesn't match other hosts. If you don't have a default host, Resin will return a `404 Not Found` for any unknown host.

The following sample configuration defines an explicit virtual hosts www.slytherin.com and a default host, each with its own root directory, access-log and a single explicit <web-app> in the `htdocs` directory. The default virtual host is configured just like a typical Apache configuration, so it can be used to upgrade an Apache/PHP site to use Quercus for security and performance.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="app-tier">
  <server id="app-a" address="192.168.1.10" port="6800">
    <http port="8080"/>
  </server>

  <development-mode-error-page/>

  <resin:import path="${__FILE__}/../app-default.xml"/>

  <host id="">
    <root-directory>/var/www</root-directory>

    <access-log path="logs/access.log"/>

    <web-app id="" root-directory="htdocs"/>
  </host>

  <host id="www.slytherin.com">
    <host-alias>slytherin.com</host-alias>

    <root-directory>/var/slytherin</root-directory>

    <access-log path="logs/access.log"/>

    <web-app id="" root-directory="htdocs"/>
  </host>

</cluster>
</resin>
```

Example: /etc/resin/resin.xml

Browsing http://gryffindor.caucho.com/test.php will look for /var/www/htdocs/test.php.

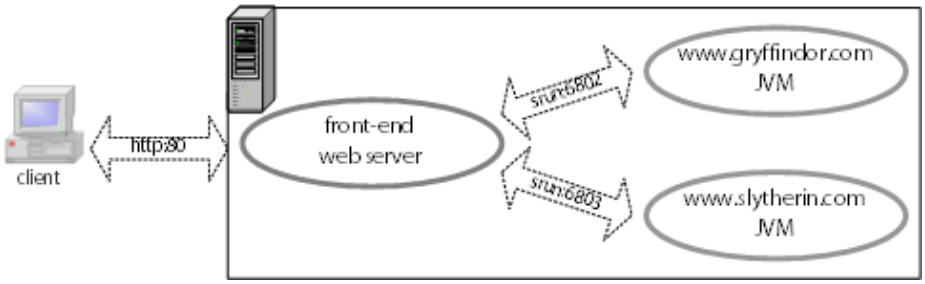
Browsing http://slytherin.caucho.com/test.php will look for /var/slytherin/htdocs/test.php.

Server per virtual host

In some ISP setups, it may make sense to assign a server for each virtual host. The isolation of web-apps may not be sufficient; each host needs a separate JVM. In this configuration, each <host> belongs to its own <cluster> and has a dedicated <server>. Normally, this configuration will operate using load-balancing, so the load-balance server will dispatch requests as appropriate.

For further security restrictions, see the watchdog section. ISPs can also use the watchdog to assign different <user-name> values for each host and can even create chroot directories for each JVM.

A front-end web server receives all requests, and is configured to dispatch to back-end Resin server that correspond to the host name.



Back-end JVMs

Each host is placed in its own <cluster> with a dedicated <server>. Since the server listens to a TCP port for load-balancing and clustering messages, each server on the maching needs a different server port.

In this example, the virtual hosts `www.gryffindor.com` and `www.slytherin.com` each get their own server. The backend clusters have their own virtual host. The frontend load-balancer dispatches the <load-balance> tags to the backend.

This example is split into two blocks to emphasize the frontend and backend. Typically, they will both actually be in the same resin.xml to ensure consistency.

```
<resin xmlns="http://caucho.com/ns/resin"
          xmlns:resin="http://caucho.com/ns/resin/core">

  <cluster-default>
    <resin:import path="${resin.home}/conf/app-default.xml"/>

    <host-default>
      <web-app-deploy path="webapps"/>
    </host-default>
  </cluster-default>

  <cluster id="gryffindor>
    <server id="gryffindor" host="localhost" port="6800"/>

    <host id="www.gryffindor.com">

      <root-directory>/var/www/gryffindor</root-directory>

    </host>
  </cluster>

  <cluster id="slytherin">
    <server id="slytherin" host="localhost" port="6801"/>

    <host id="www.slytherin.com">

      <root-directory>/var/www/slytherin</root-directory>

    </host>
  </cluster>

  <cluster id="web-tier">
    <!-- see below -->
    ...
  </cluster>

</resin>
```

Example: /etc/resin/resin.xml for backend

Each back-end server is started separately:

```
unix> java -jar lib/resin.jar -server gryffindor start
unix> java -jar lib/resin.jar -server slytherin start
```

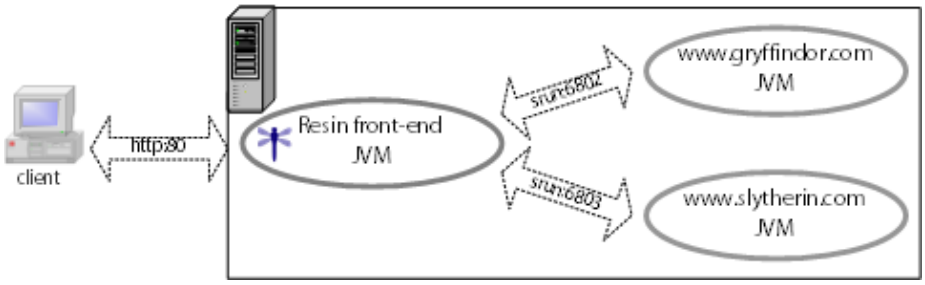
Example: starting backend servers

```
unix> java -jar lib/resin.jar -server gryffindor stop
unix> java -jar lib/resin.jar -server slytherin stop
```

Example: stopping backend servers

Resin web-tier load balancer

The host-specific back-end servers are ready to receive requests on their srun ports. A third Resin server can be used as the front-end load-balancer. It receives all requests and dispatches to the back-end servers.



The Resin web server is configured using <rewrite-dispatch> with a <load-balance directive to dispatch to the back-end server. A cluster is defined for each back-end host, so that the <load-balance> knows how to find them.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server-default>
      <http port="80"/>
    </server-default>

    <server id="web" address="192.168.2.1" port="6800"/>

    <host id="gryffindor.com">
      <web-app id="/">

        <rewrite-dispatch>
          <load-balance regexp="" cluster="gryffindor"/>
        <rewrite-dispatch>

      </web-app>
    </host>

    <host id="slytherin.com">
      <web-app id="/">

        <rewrite-dispatch>
          <load-balance regexp="" cluster="slytherin"/>
        <rewrite-dispatch>

      </web-app>
    </host>
  </cluster>

  <cluster id="gryffindor">
    <server id="gryffindor" address="192.168.2.2" port="6800"/>

    <host id="www.gryffindor.com">
      ...
    </host>
  </cluster>

  <cluster id="slytherin">
    <server id="slytherin" address="192.168.2.2" port="6801"/>

    ...
  </cluster>
</resin>
```

Example: /etc/resin/resin.xml for front-end web server

Starting the servers on Unix The front-end server JVM is started similar to the back-end JVMs:

```
unix> java -jar lib/resin.jar -server web -conf conf/resin.xml start
...
unix> java -jar lib/resin.jar -server web -conf conf/resin.xml stop
```

Example: starting the load balancer

Starting the servers on Windows With Windows, each JVM is installed as a service.

```
win> resin.exe -install-as "Resin" -server resin -conf conf/resin.xml
win> resin.exe -install-as "Resin www.gryffindor.com" -server gryffindor -conf conf/gryffindor.xml
win> resin.exe -install-as "Resin www.slytherin.com"-server slytherin -conf conf/slytherin.xml
```

Example: installing as win32 service

You will either need to reboot the machine or start the service from the Control Panel/Services panel to start the server. On a machine reboot, NT will automatically start the service.

There is a bug in many JDKs which cause the JDK to exit when the administrator logs out. JDK 1.4 and later can avoid that bug if the JDK is started with -Xrs.

Configuration tasks

host naming

The virtual host name can be configured by an explicit <host-name>, a <host-alias>, a <host-alias-regexp>, by the <host> tag or implicitly by the <host-deploy>. For explicit configuration styles, the host name and alias configuration will generally be in the resin.xml. For dynamic configuration, the host aliases will typically be in an included `host.xml` inside the host directory.

The default host catches all unmatches hosts. Simpler sites will use the default host for all requests, while security-conscious sites may remove the default host entirely. If the default host is not configured, Resin will return a `404 Not Found` .

host.xml

The `host.xml` is an optional file where virtual hosts can put host-common configuration. The `host.xml` is a good place for shared resources like authentication, database pools or host-wide beans and services. It's also a location for the <host-alias> in a dynamic hosting configuration.

The `host.xml` is configured in a <host-deploy> or <host-default> by adding a <resin:import> tag specifying the host.xml name and location. Because the <host-default> applies the <resin:import> to every virtual host, it becomes a common system-wide configuration file.

web-applications

Hosts must define web-apps in order to serve files, servlets, or PHP pages. If the host is missing all webapps, Resin will return `404 Not Found` for all requests make to the host.

Both explicit <web-app> and dynamic web-app-deploy tags are used to configure webapps. The explicit style is generally used for Apache-style configuration, while the dynamic style is generally used for Java app-server .war configuration.

Remember, Resin's default servlets like the file, JSP, and PHP servlets also need to be defined before they're used. So all Resin configuration files need to have a <resin:import> of the `conf/app-default.xml` configuration file either in the <cluster> or in a shared <cluster-default>. If the app-default.xml is missing, Resin will not serve static files, JSP, or PHP, and will not even look in the WEB-INF for resin-web.xml, classes, or lib.

IP-Based Virtual Hosting

While Resin's virtual hosting is primarily aimed at named-based virtual hosts, it's possible to run Resin with IP-Based virtual hosts.

With IP virtual hosting, each <http> block is configured with the virtual host name. This configuration will override any virtual host supplied by the browser.

```
<resin xmlns="http://caucho.com/ns/resin">

<cluster id="web-tier">
  <server id="a">

    <http address="192.168.0.1" port="80"
          virtual-host="slytherin.caucho.com"/>

    <http address="192.168.0.2" port="80"
          virtual-host="gryffindor.caucho.com"/>

  </server>

  ...

  <host id="slytherin.caucho.com">
    ...
  </host>
</cluster>
</resin>
```

Internationalization

Resin's virtual hosting understands host names encoded using rfc3490 (Internationalizing Domain Names in Applications). This support should be transpar-

ent. Just specify the virtual host as usual, and Resin will translate the brower's encoded host name the unicode string.

Support, of course, depends on the browser. Mozilla 1.4 supports the encoding.

Virtual Hosts with Apache or IIS

A common configuration uses virtual hosts with Apache or IIS. As usual, Apache or IIS will pass matching requests to Resin.

Apache

The Resin JVM configuration with Apache is identical to the standalone configuration. That similarity makes it easy to debug the Apache configuration by retreating to Resin standalone if needed.

The `ServerName` directive in Apache is vital to make Resin's virtual hosting work. When Apache passes the request to Resin, it tells Resin the `ServerName` . Without the `ServerName` , Resin can get very confused which host to serve.

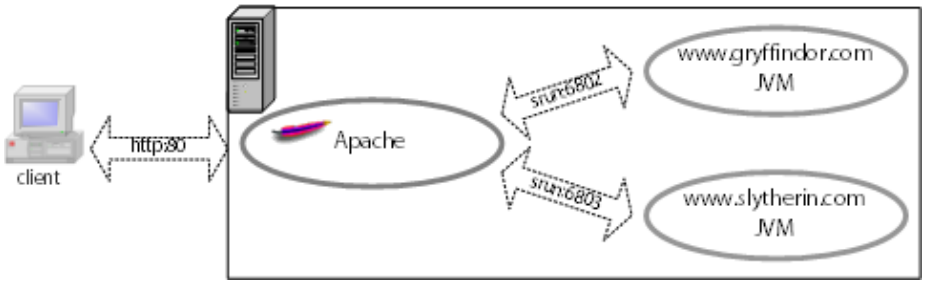
```
LoadModule caucho_module /usr/local/apache/libexec/mod_caucho.so

ResinConfigServer localhost 6802

<VirtualHost 127.0.0.1>
  ServerName gryffindor.caucho.com
</VirtualHost>

<VirtualHost 192.168.0.1>
  ServerName slytherin.caucho.com
</VirtualHost>
```

httpd.conf

Note: You'll the LoadModule must appear before the ResinConfigServer for Apache to properly understand the ResinConfigServer command. If they're missing, Apache will send an error.

Apache front-end

The host-specific back-end JVMs are ready to receive requests on their srun ports. Apache is the front-end server, and is configured to dispatch to the appropriate back-end Resin JVM for the host:




```
<VirtualHost 127.0.0.1>
  ServerName gryffindor.caucho.com
  ResinConfigServer 192.168.0.10 6800
</VirtualHost>

<VirtualHost 192.168.0.1>
  ServerName slytherin.caucho.com
  ResinConfigServer 192.168.0.11 6800
</VirtualHost>
```

httpd.conf

When you restart the Apache web server, you can look at `http://gryffindor/caucho-status` and `http://slytherin/caucho-status` to check your configuration. Check that each virtual host is using the server `address` and `port` that you expect.

IIS

Configuration and installation for IIS virtual sites is discussed in the IIS installation section.

Testing virtual hosts

During development and testing, it is often inconvenient or impossible to use real virtual host names that are registered as internet sites, and resolve to an internet-available IP address. OS-level features on the test client machine can be used to map a virtual host name to an IP address.

For example, developers often run the Resin server and the test client (usually a browser) on the same machine. The OS is configured to map the "www.gryffindor.com" and "www.slytherin.com" names to "127.0.0.1", pointing these host names back to computer that the client is running on.

Unix user's edit the file `/etc/hosts` :

```
127.0.0.1       localhost

127.0.0.1       www.gryffindor.com
127.0.0.1       www.slytherin.com
```

/etc/hosts

Windows user edit the file `C:\WINDOWS\SYSTEM32\DRIVERS\ETC\HOSTS` :

```
127.0.0.1       localhost

127.0.0.1       www.gryffindor.com
127.0.0.1       www.slytherin.com
```

C:\WINDOWS\SYSTEM32\DRIVERS\ETC\HOSTS

Chapter 7

Clustering

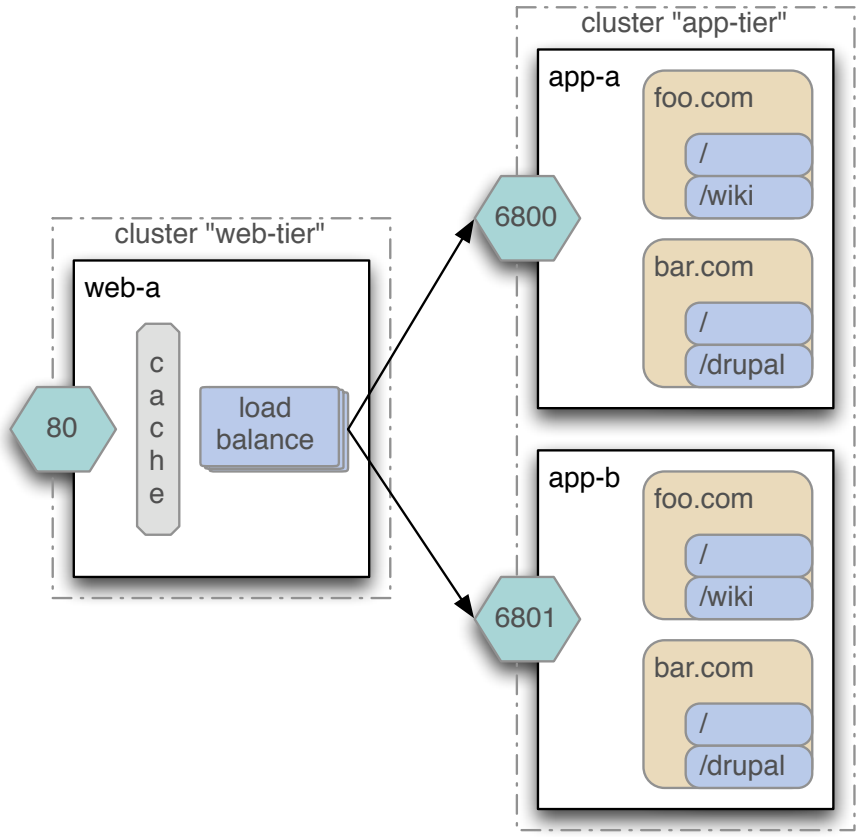
7.1 Resin Clustering

Load Balancing

Resin Professional includes a <load-balance> rewrite tag that can balance requests to an application-tier cluster of backend servers. Because it is implemented as a servlet, this configuration is the most flexible. A site might use 192.168.0.1 as the frontend load balancer, and send all requests for /foo to the backend host 192.168.0.10 and all requests to /bar to the backend host 192.168.0.11. Since Resin has an integrated HTTP proxy cache, the web-tier machine can cache results for the backend servers.

Load balancing divides the Resin servers into two clusters: the web-tier and the app-tier. In Resin 3.2, all the cluster and load balance configuration is in a single resin.xml. The actual deployed server is selected with the "-server web-a" command-line argument.

Using Resin as the load balancing web server requires a minimum of two clusters: one for the load balancing server, and one for the backend servers. The front cluster will dispatch to the backend servers, while the backend will actually serve the requests. Both clusters can be defined in the same resin.xml file.



The web-tier caches and load balances

In the following example, there are three servers and two clusters. The first server "web-a" (192.168.0.1), which is in cluster "web-tier", is the load balancer. It has an <http> listener, receives requests from browsers, and dispatches them to the backend servers "app-a" and "app-b" (192.168.0.10 and 192.168.0.11).

```
<resin xmlns="http://caucho.com/ns/resin">

<cluster id="web-tier">
  <server-default>
    <http port="80"/>
  </server-default>

  <server id="web-a" address="192.168.0.1" port="6800"/>

  <cache disk-size="1024M" memory-size="256M"/>

  <host id="">
    <web-app id="/">
      <!-- balance all requests to cluster app-tier -->
      <rewrite-dispatch>
        <load-balance regexp="" cluster="app-tier"/>
      </rewrite-dispatch>
    </web-app>
  </host>
</cluster>

<cluster id="app-tier">
  <server id="app-a" address="192.168.0.10" port="6800"/>
  <server id="app-b" address="192.168.0.11" port="6800"/>

  <persistent-store type="cluster">
    <init path="cluster"/>
  </persistent-store>

  <web-app-default>
    <session-config>
      <use-persistent-store/>
    </session-config>
  </web-app-default>

  <host id="www.foo.com">
    ...
  </host>
</cluster>

</resin>
```

Example: resin.xml for load balancing

The LoadBalanceServlet selects a backend server using a round-robin policy. Although the round-robin policy is simple, in practice it is as effective as complicated balancing policies. In addition, because it's simple, round-robin is more robust and faster than adaptive policies.

The app-tier processes requests

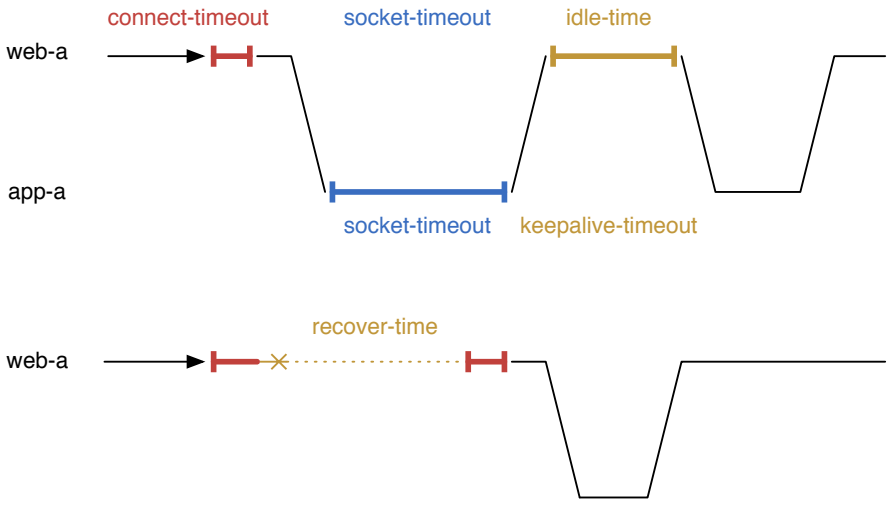
The backend servers belong to the "app-tier" cluster. In this case, there are two backend servers, "app-a" and "app-b". The server names must be

unique and also the <address,port> pair must be unique because TCP will not allow two processes to listen to exactly the same port and address combination. If you have two backend servers on the same machine, they must have different ports, like 6801 and 6800. In other words, you can have <192.168.0.10,6800> and <192.168.0.11,6800> on separate machines or you can have <192.168.0.10,6800> and <192.168.0.10,6801> on the same machine.

Because the backend cluster all serves the same content, the <host> and <web-app> configuration is identical. Only JVM-specific configuration like ports, addresses, and memory configuration belongs to the server.

Sites using sessions will often configure distributed sessions to make that the load balancer's failover will ensure a smooth switch of sessions from one backend to another.

Timeouts



- **load-balance-connect-timeout**: the load balancer timeout for the `connect()` system call to complete to the app-tier (5s).
- **load-balance-idle-time**: load balancer timeout for an idle socket before reusing it (5s).
- **load-balance-recover-time**: the load balancer connection failure wait time before trying a new connection (15s).
- **load-balance-socket-timeout**: the load balancer timeout for a valid request to complete (665s).
- **keepalive-timeout**: the app-tier timeout for a keepalive connection (15s)
- **socket-timeout**: the app-tier timeout for a read or write (65s)

Starting the servers

```
192.168.0.1> java -jar lib/resin.jar -server web-a start

192.168.0.10> java -jar lib/resin.jar -server app-a start
192.168.0.11> java -jar lib/resin.jar -server app-b start
```

Starting each server

Dispatching

In most cases, the web-tier will dispatch everything to the app-tier servers. Because of Resin's proxy cache, the web-tier servers will serve static pages as fast as if they were local pages.

In some cases, though, it may be important to send different requests to different backend clusters. The <load-balance> tag can choose clusters based on URL patterns.

The following <rewrite-dispatch> keeps all *.png, *.gif, and *.jpg files on the web-tier, sends everything in /foo/* to the foo-tier cluster, everything in /bar/* to the bar-tier cluster, and keeps anything else on the web-tier.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server id="web-a">
      <http port="80"/>
    </server>

    <cache memory-size="64m"/>

    <host id="">
      <web-app id="/">

        <rewrite-dispatch>
          <dispatch regexp="(\.png|\.gif|\.jpg)"/>

          <load-balance regexp="^/foo" cluster="foo-tier"/>

          <load-balance regexp="^/bar" cluster="bar-tier"/>
        </rewrite-dispatch>

      </web-app>
    </host>
  </cluster>

  <cluster id="foo-tier">
    ...
  </cluster>

  <cluster id="bar-tier">
    ...
  </cluster>
</resin>
```

split dispatching

Persistent Sessions

A session needs to stay on the same JVM that started it. Otherwise, each JVM would only see every second or third request and get confused.

To make sure that sessions stay on the same JVM, Resin encodes the cookie with the host number. In the previous example, the hosts would generate cookies like:

INDEX	COOKIE PREFIX
1	a xxx
2	b xxx
3	c xxx

On the web-tier, Resin will decode the cookie and send it to the appropriate

host. So `bacX8ZwooOz` would go to app-b.

In the infrequent case that app-b fails, Resin will send the request to app-a. The user might lose the session but that's a minor problem compared to showing a connection failure error. To save sessions, you'll need to use distributed sessions. Also take a look at tcp sessions.

The following example is a typical configuration for a distributed server using an external hardware load-balancer, i.e. where each Resin is acting as the HTTP server. Each server will be started as `-server a` or `-server b` to grab its specific configuration.

In this example, sessions will only be stored when the server shuts down, either for maintenance or with a new version of the server. This is the most lightweight configuration, and doesn't affect performance significantly. If the hardware or the JVM crashes, however, the sessions will be lost. (If you want to save sessions for hardware or JVM crashes, remove the <save-only-on-shutdown/> flag.)

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="app-tier">
  <server-default>
    <http port='80'/>
  </server-default>

  <server id='app-a' address='192.168.0.1'/>
  <server id='app-b' address='192.168.0.2'/>
  <server id='app-c' address='192.168.0.3'/>

  <persistent-store type="cluster">
    <init path="cluster"/>
  </persistent-store>

  <web-app-default>
    <!-- enable tcp-store for all hosts/web-apps -->
    <session-config>
      <use-persistent-store/>
      <save-only-on-shutdown/>
    </session-config>
  </web-app-default>

  ...
</cluster>
</resin>
```

resin.xml

Choosing a backend server

Requests can be made to specific servers in the app-tier. The web-tier uses the value of the jsessionid to maintain sticky sessions. You can include an explicit jsessionid to force the web-tier to use a particular server in the app-tier.

Resin uses the first character of the jsessionid to identify the backend server

to use, starting with 'a' as the first backend server. If wwww.example.com resolves to your web-tier, then you can use:

1. http://www.example.com/proxooladmin;jsessionid=abc
2. http://www.example.com/proxooladmin;jsessionid=bcd
3. http://www.example.com/proxooladmin;jsessionid=cde
4. http://www.example.com/proxooladmin;jsessionid=def
5. http://www.example.com/proxooladmin;jsessionid=efg
6. etc.

<persistent-store>

Configuration for persistent store uses the persistent-store tag.

File Based

- For single-server configurations
- Useful in development when classes change often

Persistent sessions are configured in the `web-app` . File-based sessions use `file-store` .

```
<web-app xmlns="http://caucho.com/ns/resin">
  <session-config>
    <file-store>WEB-INF/sessions</file-store>
  </session-config>
</web-app>
```

Sessions are stored as files in the `file-store` directory. When the session changes, the updates will be written to the file. After Resin loads an Application, it will load the stored sessions.

File-based persistence is not useful in multi-server environments. Although a network filesystem such as NFS will allow all the servers to access the same filesystem, it's not designed for the fine-grained access. For example, NFS will cache pages. So if one server modifies the page, e.g. a session value, the other servers may not see the change for several seconds.

Distributed Sessions

Distributed sessions are intrinsically more complicated than single-server sessions. Single-server session can be implemented as a simple memory-based Hashtable. Distributed sessions must communicate between machines to ensure the session state remains consistent.

Load balancing with multiple machines either uses `sticky sessions` or `symmetrical sessions` . Sticky sessions put more intelligence on the load balancer, and symmetrical sessions puts more intelligence on the JVMs. The choice of which to use depends on what kind of hardware you have, how many machines you're using and how you use sessions.

Distributed sessions can use a database as a backing store, or they can distribute the backup among all the servers using TCP.

Symmetrical Sessions Symmetrical sessions happen with dumb load balancers like DNS round-robin. A single session may bounce from machine A to machine B and back to machine B. For JDBC sessions, the symmetrical session case needs the `always-load-session` attribute described below. Each request must load the most up-to-date version of the session.

Distributed sessions in a symmetrical environment are required to make sessions work at all. Otherwise the state will end up spread across the JVMs. However, because each request must update its session information, it is less efficient than sticky sessions.

Sticky Sessions Sticky sessions require more intelligence on the load-balancer, but are easier for the JVM. Once a session starts, the load-balancer will always send it to the same JVM. Resin's load balancing, for example, encodes the session id as 'aaaXXX' and 'baaXXX'. The 'aaa' session will always go to JVM-a and 'baa' will always go to JVM-b.

Distributed sessions with a sticky session environment add reliability. If JVM-a goes down, JVM-b can pick up the session without the user noticing any change. In addition, distributed sticky sessions are more efficient. The distributor only needs to update sessions when they change. So if you update the session once when the user logs in, the distributed sessions can be very efficient.

always-load-session Symmetrical sessions must use the 'always-load-session' flag to update each session data on each request. always-load-session is only needed for jdbc-store sessions. tcp-store sessions use a more-sophisticated protocol that eliminates the need for always-load-session, so tcp-store ignores the always-load-session flag.

The `always-load-session` attribute forces sessions to check the store for each request. By default, sessions are only loaded from persistent store when they are created. In a configuration with multiple symmetric web servers, sessions can be loaded on each request to ensure consistency.

always-save-session By default, Resin only saves session data when you add new values to the session object, i.e. if the request calls `setAttribute` . This may be insufficient when storing large objects. For example, if you change an internal field of a large object, Resin will not automatically detect that change and will not save the session object.

With `always-save-session` Resin will always write the session to the store at the end of each request. Although this is less efficient, it guarantees that updates will get stored in the backup after each request.

Database Based

Database backed sessions are the easiest to understand. Session data gets serialized and stored in a database. The data is loaded on the next request.

For efficiency, the owning JVM keeps a cache of the session value, so it only needs to query the database when the session changes. If another JVM stores a new session value, it will notify the owner of the change so the owner can update its cache. Because of this notification, the database store is cluster-aware.

In some cases, the database can become a bottleneck. By adding load to an already-loaded system, you may harm performance. One way around that bottleneck is to use a small, quick database like MySQL for your session store and save the "Big Iron" database like Oracle for your core database needs.

The database must be specified using a <`database`> . The database store will automatically create a `session` table.

The JDBC store needs to know about the other servers in the cluster in order to efficiently update them when changes occur to the server.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="app-tier">
  <server-default>
    <http port="80"/>
  </server-default>

  <server id="app-a" address="192.168.2.10" port="6800"/>
  <server id="app-b" address="192.168.2.11" port="6800"/>

  <database jndi-name="jdbc/session">
    ...
  </database>

  <persistent-store type="jdbc">
    <init>
      <data-source>jdbc/session<data-source>
    </init>
  </persistent-store>
  ...

  <web-app-default>
    <session-config>
      <use-persistent-store/>
    </session-config>
  </web-app-default>
  ...
</cluster>
</resin>
```

JDBC store

The persistent store is configured in the <server> with persistent-store. Each web-app which needs distributed sessions must enable the persistent store with a use-persistent-store tag in the session-config.

data-source	data source name for the table
table-name	database table for the session data
blob-type	database type for a blob
max-idle-time	cleanup time

```
CREATE TABLE persistent_session (
  id VARCHAR(64) NOT NULL,
  data BLOB,
  access_time int(11),
  expire_interval int(11),
  PRIMARY KEY(id)
)
```

The store is enabled with <use-persistent-store> in the session config.

```
<web-app xmlns="http://caucho.com/ns/resin">
  <session-config>
    <use-persistent-store/>
    <always-save-session/>
  </session-config>
</web-app>
```

Cluster Sessions

The distributed cluster stores the sessions across the cluster servers. In some configurations, the cluster store may be more efficient than the database store, in others the database store will be more efficient.

With cluster sessions, each session has an owning JVM and a backup JVM. The session is always stored in both the owning JVM and the backup JVM.

The cluster store is configured in the in the <cluster>. It uses the <server> hosts in the <cluster> to distribute the sessions. The session store is enabled in the <session-config> with the <use-persistent-store>.

```
<resin xmlns="http://caucho.com/ns/resin">
  ...

  <cluster id="app-tier">
    <server id="app-a" host="192.168.0.1" port="6802"/>
    <server id="app-b" host="192.168.0.2" port="6802"/>

    <persistent-store type="cluster">
      <init path="cluster"/>
    </persistent-store>

    ...
  </cluster>
</resin>
```

The configuration is enabled in the `web-app` .

```
<web-app xmlns="http://caucho.com/ns/resin">
  <session-config>
    <use-persistent-store="true"/>
  </session-config>
</web-app>
```

The <srun> and <srun-backup> hosts are treated as a cluster of hosts. Each host uses the other hosts as a backup. When the session changes, the updates will be sent to the backup host. When the host starts, it looks up old sessions in the other hosts to update its own version of the persistent store.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="app-tier">

  <server-default>
    <http port='80'/>
  </server-default>

  <server id="app-a" address="192.168.2.10" port="6802"/>
  <server id="app-b" address="192.168.2.11" port="6803"/>

  <persistent-store type="cluster">
    <init path="cluster"/>
  </persistent-store>

  <host id=''>
  <web-app id=''>

  <session-config>
    <use-persistent-store="true"/>
  </session-config>

  </web-app>
  </host>
</cluster>
</resin>
```

Symmetric load-balanced servers

Clustered Distributed Sessions

Resin's cluster protocol for distributed sessions can is an alternative to JDBC-based distributed sessions. In some configurations, the cluster-stored sessions will be more efficient than JDBC-based sessions. Because sessions are always duplicated on separate servers, cluster sessions do not have a single point of failure. As the number of servers increases, JDBC-based sessions can start overloading the backing database. With clustered sessions, each additional server shares the backup load, so the main scalability issue reduces to network bandwidth. Like the JDBC-based sessions, the cluster store sessions uses sticky-session caching to avoid unnecessary network traffic.

Configuration

The cluster configuration must tell each host the servers in the cluster and it must enable the persistent in the session configuration with use-persistent-store. Because session configuration is specific to a virtual host and a web-application, each web-app needs `use-persistent-store` enabled individually. The web-app-default tag can be used to enable distributed sessions across an entire site.

Most sites using Resin's load balancing will already have the cluster <`srun`> configured. Each <`srun`> block corresponds to a host, including the current

host. Since cluster sessions uses Resin's srun protocol, each host must listen for srun requests.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">

    <server id="app-a" host="192.168.0.1"/>
    <server id="app-b" host="192.168.0.2"/>
    <server id="app-c" host="192.168.0.3"/>
    <server id="app-d" host="192.168.0.4"/>

    <persistent-store type="cluster">
      <init path="cluster"/>
    </persistent-store>

    ...
    <host id="">
    <web-app id='myapp'>
      ...
      <session-config>
        <use-persistent-store/>
      </session-config>
    </web-app>
    </host>
  </cluster>
</resin>
```

resin.xml fragment

Usually, hosts will share the same resin.xml. Each host will be started with a different `-server xx` to select the correct block. On Unix, startup will look like:

```
resin-3.2.x> bin/resin.sh -conf conf/resin.xml -server c start
```

Starting HostC on Unix

On Windows, Resin will generally be configured as a service:

```
resin-3.2.x> bin/resin -conf conf/resin.xml -server c -install-as ResinC
```

Starting HostC on Windows

always-save-session Resin's distributed sessions needs to know when a session has changed in order to save the new session value. Although Resin can detect when an application calls `HttpSession.setAttribute` , it can't tell if an internal session value has changed. The following Counter class shows the issue:

```
package test;

public class Counter implements java.io.Serializable {
  private int _count;

  public int nextCount() { return _count++; }
}
```

Counter.java

Assuming a copy of the Counter is saved as a session attribute, Resin doesn't know if the application has called `nextCount` . If it can't detect a change, Resin will not backup the new session, unless `always-save-session` is set. When `always-save-session` is true, Resin will back up the session on every request.

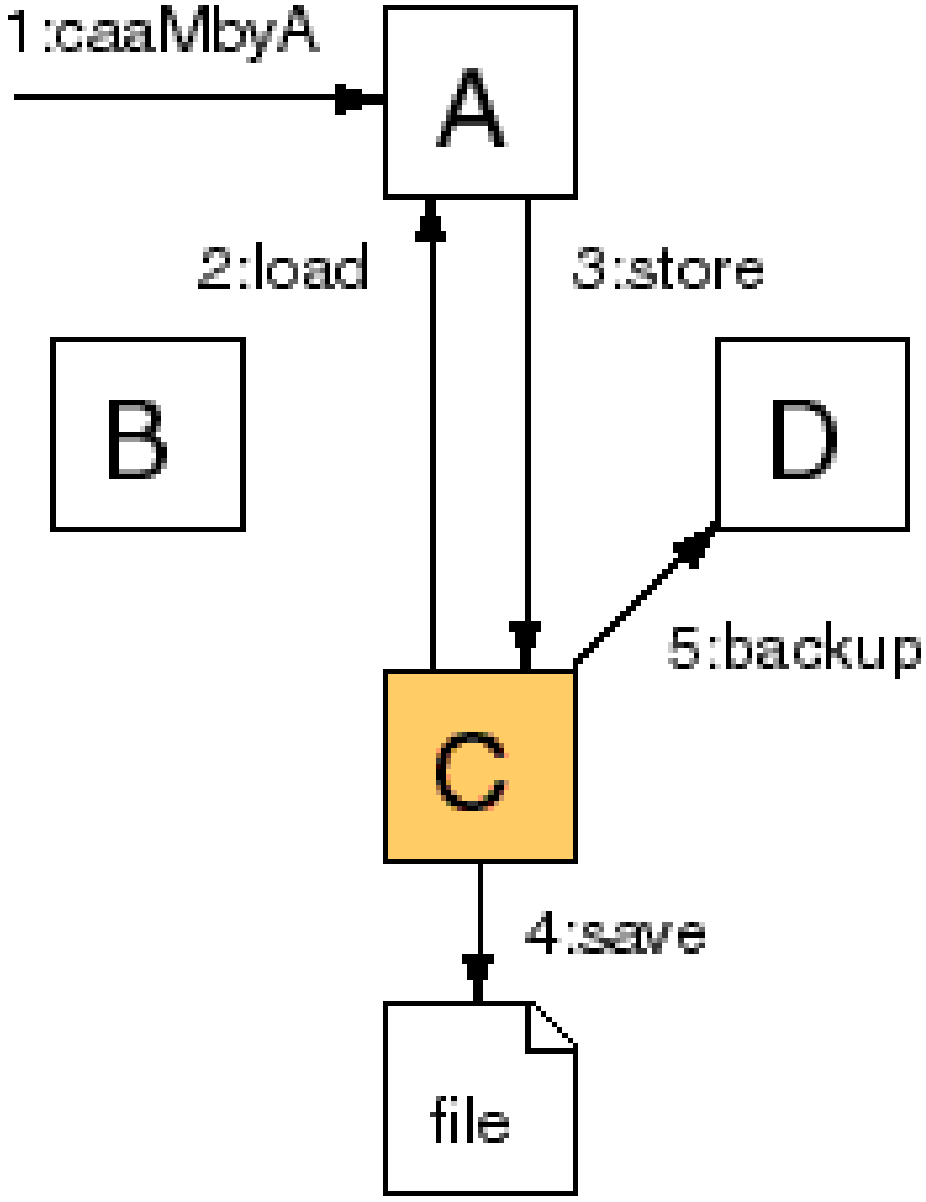
```
...
<web-app id="/foo">
...
<session-config>
  <use-persistent-store/>
  <always-save-session/>
</session-config>
...
</web-app>
```

Like the JDBC-based sessions, Resin will ignore the `always-load-session` flag for cluster sessions. Because the cluster protocol notifies servers of changes, `always-load-session` is not needed.

Serialization Resin's distributed sessions relies on Java serialization to save and restore sessions. Application object must `implement java.io.Serializable` for distributed sessions to work.

Protocol Examples

Session Request To see how cluster sessions work, consider a case where the load balancer sends the request to a random host. HostC owns the session but the load balancer gives the request to HostA. In the following figure, the request modifies the session so it must be saved as well as loaded.

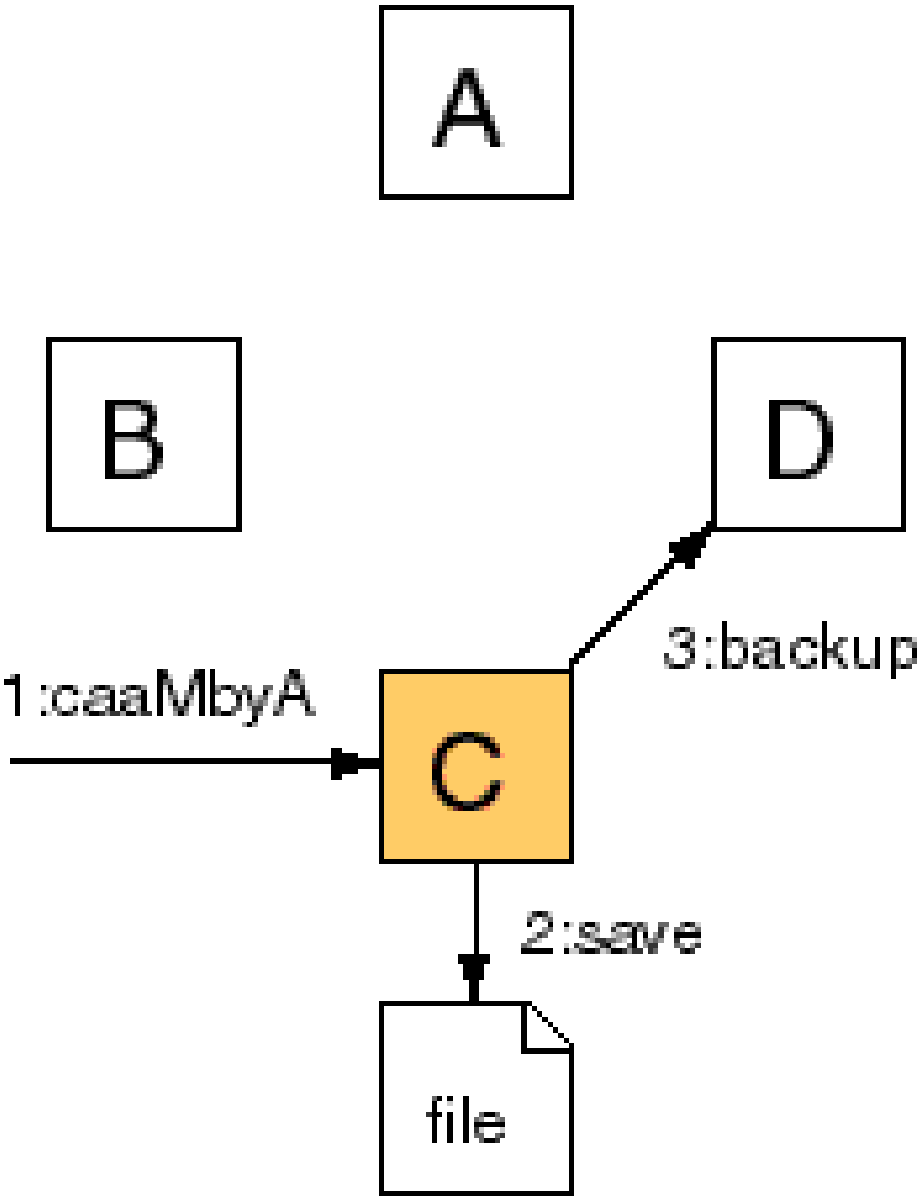

1:caaMbyA
A
2:load
3:store
B
D
5:backup
C
4:save
file

The session id encodes the owning host. The example session id, `ca8MbyA` , decodes to an srun-index of 3, mapping to HostC. Resin determines the backup host from the cookie as well. HostA must know the owning host for every cookie so it can communicate with the owning srun. The example configuration defines all the sruns HostA needs to know about. If HostC is unavailable, HostA can use its configuration knowledge to use HostD as a backup for `ca8MbyA` instead..

When the request first accesses the session, HostA asks HostC for the serialized session data (`2:load`). Since HostA doesn't cache the session data, it must ask HostC for an update on each request. For requests that only read the session, this TCP load is the only extra overhead, i.e. they can skip `3-5` . The `always-save-session` flag, in contrast, will always force a write.

At the end of the request, HostA writes any session updates to HostC (`3:store`). If always-save-session is false and the session doesn't change, this step can be skipped. HostA sends the new serialized session contents to HostC. HostC saves the session on its local disk (`4:save`) and saves a backup to HostD (`5:backup`).

Sticky Session Request Smart load balancers that implement sticky sessions can improve cluster performance. In the previous request, Resin's cluster sessions maintain consistency for dumb load balancers or twisted clients like the AOL browsers. The cost is the additional network traffic for `2:load` and `3:store` . Smart load-balancers can avoid the network traffic of `2` and `3` .


A
B
D
3:backup
1:caaMbyA
C
2:save
file

HostC decodes the session id, `caaMbyA` . Since it owns the session, HostC gives the session to the servlet with no work and no network traffic. For a read-only request, there's zero overhead for cluster sessions. So even a semi-intelligent load balancer will gain a performance advantage. Normal browsers will have zero overhead, and bogus AOL browsers will have the non-sticky session overhead.

A session write saves the new serialized session to disk (`2:save`) and to HostD (`3:backup`). `always-save-session` will determine if Resin can take advantage of read-only sessions or must save the session on each request.

Disk copy Resin stores a disk copy of the session information, in the location specified by the `path` . The disk copy serves two purposes. The first is that it allows Resin to keep session information for a large number of sessions. An efficient memory cache keeps the most active sessions in memory and the disk holds all of the sessions without requiring large amounts of memory. The second purpose of the disk copy is that the sessions are recovered from disk when the server is restarted.

Failover Since the session always has a current copy on two servers, the load balancer can direct requests to the next server in the ring. The backup server is always ready to take control. The failover will succeed even for dumb load balancers, as in the non-sticky-session case, because the srun hosts will use the backup as the new owning server.

In the example, either HostC or HostD can stop and the sessions will use the backup. Of course, the failover will work for scheduled downtime as well as server crashes. A site could upgrade one server at a time with no observable downtime.

Recovery When HostC restarts, possibly with an upgraded version of Resin, it needs to use the most up-to-date version of the session; its file-saved session will probably be obsolete. When a "new" session arrives, HostC loads the saved session from both the file and from HostD. It will use the newest session as the current value. Once it's loaded the "new" session, it will remain consistent as if the server had never stopped.

No Distributed Locking Resin's cluster sessions does not lock sessions. For browser-based sessions, only one request will execute at a time. Since browser sessions have no concurrently, there's no need for distributed locking. However, it's a good idea to be aware of the lack of distributed locking.

See Also

- Distributed Sessions
- Distributed Sessions with Cluster Store
- <persistent-store>

Chapter 8

Web Applications

8.1 An Overview of Web Applications

Web applications

Each web application is part of the web server. It has a unique name or path that identifies it within the server.

Each web application has a corresponding url. The url begins with the part needed to identify the server, followed by the webapp path: `http://server/webapp-name` . Each server has one web-app that is the default, it is the one that is used when no webapp-name is provided.

Web applications are "deployed" within a web server, such as Resin. The simplest way to "deploy" a new web application is to the create a subdirectory in `$RESIN_HOME/webapps/ webapp-name` . The special webapp name `ROOT` is used for the default web application. (There are other deployment options, but for the purposes of this discussion the one described here is used).

A web application has "web components", such as Servlets, Filters, JSP's, supporting Java source files, and supporting java libraries.

Try it!

You can make your own web application in a local install of Resin. Make a directory `$RESIN_HOME/webapps/test` . Use the url `http://localhost:8080/test` to access the web application.

To start with, you can make a file named `$RESIN_HOME/test/index.jsp` .

```
Hello, world!
```

`$RESIN_HOME/test/index.jsp`

`index.jsp` is a JSP file, and is also the name of the default page to show for a directory. So you can use the url `http://localhost:8080/test/index.jsp`

in your browser, or since `index.jsp` is the default page to show, you can use `http://localhost:8080/test` .

Example web application

For example, `www.hogwarts.com` has two web applications, the `default` web application, and a web-application named "intranet".

Server: www.hogwarts.com
Server URL: http://www.hogwarts.com

webapp: default webapp
webapp URL: http://www.hogwarts.com/
filesystem directory: $RESIN_HOME/webapps/ROOT
default jsp page: $RESIN_HOME/webapps/ROOT/index.jsp

webapp: intranet
webapp URL: http://www.hogwarts.com/intranet
filesystem directory: $RESIN_HOME/webapps/intranet
default jsp page: $RESIN_HOME/webapps/intranet/index.jsp

Components of a web application

Servlet

From the Servlet Specification 2.2:

> A servlet is a web component, managed by a container, that generates dynamic content. Servlets are small, platform independent Java classes compiled to an architecture neutral bytecode that can be loaded dynamically into and run by a web server. Servlets interact with web clients via a request response paradigm implemented by the servlet container. This request-response model is based on the behavior of the Hypertext Transfer Protocol (HTTP).

A Servlet is a Java class that has a method that gets called with information about a client request and is expected to produce some kind of result to be sent back to the client. It is just like any other class in Java, it happens to inherit from , so Resin can call certain methods on it when a request is made.

A Servlet class is made available by placing the `.java` source file in the approriate sub-directory and file of `WEB-INF/classes` :

```
package example;

import java.io.*;

import javax.servlet.http.*;
import javax.servlet.*;

/**
 * Hello world servlet.  Most servlets will extend
 * javax.servlet.http.HttpServlet as this one does.
 */
public class HelloServlet extends HttpServlet {
  /**
   * Initialize the servlet.  Servlets should override this method
   * if they need any initialization like opening pooled
   * database connections.
   */
  public void init() throws ServletException
  {
  }

  /**
   * Implements the HTTP GET method.  The GET method is the standard
   * browser method.
   *
   * @param request the request object, containing data from the browser
   * @param repsonse the response object to send data to the browser
   */
  public void doGet (HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException
  {
    // Returns a writer to write to the client
    PrintWriter out = response.getWriter();

    // Write a string to the browser.
    out.println("Hello, world!");
    out.close();
  }
}
```

WEB-INF/classes/example/HelloWorldServlet.java

Entries in ithe `WEB-INF/web.xml` file tell Resin the URL that should invoke the Servlet:

```
<web-app xmlns="http://caucho.com/ns/resin">
  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>example.HelloServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <url-pattern>/hello</url-pattern>
    <servlet-name>hello</servlet-name>
  </servlet-mapping>
</web-app>
```

WEB-INF/web.xml

In a web-app named "foo" on a server named "localhost" listening on port "80", the servlet is now invoked with the URL `http://localhost:8080/foo/hello` .

More information on the usage of Servlets is available in the Servlet section of the Resin documenation.

JSP

Java Server Pages are text files that contain text to be output (usually HTML or somesuch) and special directives, actions, scripting elements, and expressionsthat are used to generate results dynamically.

From the JSP 2.0 specification:

> JavaServer Pages technology supports scripting elements as well as actions. Actions encapsulate useful functionality in a convenient form that can be manipulated by tools. Expressions are used to access data. Scripts can be used to glue together this functionality in a per-page manner.

With JSP the developer specifies the content mostly as the kind of thing they want to send back to the client or browser, for example HTML. Optionally interspersed with the HTML are special xml tags (directives and actions), EL expressions, or specially marked scripting code (Java code). The special xml tags, EL expressions and Java code are used to generate dynamic ouput.

JSP's are translated into Servlets It is helpful to understand what it is that Resin does with a JSP page. Basically, it takes the JSP page and turns it into the Java code for a Servlet, a process of `translation` .

From the JSP specification:

> JSP pages are textual components. They go through two phases: a translation phase, and a request phase. Translation is done once per page. The request phase is done once per request.

The translation phase occurs when Resin takes a look at the JSP page, reads it in, and creates a Servlet. This only needs to be done once. Now when a request from a client comes in, Resin will call the appropriate method in the Servlet that it created from the.

During translation, Resin takes all of the code that has been specially marked in the JSP as java code and inserts it directly into the code for a Servlet. It takes all of the template text and makes the equivalent of print statements to generate that ouput.

Because JSP files are translated into Servlets, JSP is an extension to Java Servlets. Everything that applies to Java Servlets also applies to JSP. Much information that is relevent to JSP programming is found in documentation about Java Servlets. So you want to have the Servlet Specification around as a handy reference, as well as the JSP Specification. Any reference to the capabilities and resources available to a Servlet are also available to a JSP page.

This process is invisible to the JSP developer, all the developer needs to do is make the JSP page and Resin will look at it and turn it into a Servlet.

The syntax of a JSP file JSP has it's own section in the Resin documentation, the following is an introductory guide.**template data - The text to be output**

Unless specially marked, the text in the JSP file will be sent exactly as it is in the text file as part of the response. This is called `template data` in the JSP specification. **JSP EL and JSTL**

JSP EL is the JSP `Expression Language` . It is used to evaluate expressions that do not have side-effects (side-effects are changes to Objects). The use of EL is recognizable by it's syntax: `${'${'} expr }` .

JSTL is the JavaServer Pages Standard Tag Libray, a set of `tags` that are used to create dynamic output from JSP. These tags look like regular XML tags, and are interpreted by Resin at translation time to generate java code that performs the desired action.

EL and JSTL are used throughout this discussion, the Resin JSP documentation, the JSTL documentation provide more information.

```
<%@page session="false" contentType="text/html" %>
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c" %>

<head>
<title>A simple thing</title>
</head>
<body>

<!-- this comment gets all the way to the browser -->
<%-- this comment gets discarded when the JSP is translated into a Servlet --%>
<%
// some java code that makes the variable 'x' available to EL
pageContext.setAttribute("x",new Integer(5));
%>

The value of x is ${'${'} x }
The value of x + 2 is ${'${'} x + 2 }
Is x
less than 6?
<c:if test="${'${'} x < 6 }">
<%@include file="y.jsp" %>
</c:if>
</body>
```

x.jsp - Example JSP file using EL and JSTL

```
Yes, it is true that x is less than 6,
with a value of ${'${'} x }
```

y.jsp - Example java code in JSP file to be included

```
<head>
<title>A simple thing</title>
</head>
<body>
<!-- this comment gets all the way to the browser -->


The value of x is 5.
The value of x + 2 is 7.
Is x
less than 6?

Yes, it is true that x is less than 6,
with a value of 5.
</body>
```

x.jsp output - The result of a call to x.jsp

Prior to the introduction of JSTL and EL, JSP pages used the direct insertion of Java code to accomplish the same thing. The use of JSTL and EL is much cleaner and more maintable. **Including other files**

Often it is desirable to include the contents of another file into a JSP file. For example, sometimes there is code that you find yourself using over and over again. The mechanism for this is the `include` directive:

```
<%@ include file=" \texttt{relativeURLspec} "%>
```

Using the include directive it is exactly the same as if the included text was in the original file. The text is included at translation time - when the JSP is turned into a servlet. **Specifying content type**

A JSP page can use the `contentType` attribute of the `page` directive to indicate the content type of the response it is sending. For example, `text/html` and `text/wml` are valid content types.

Since this value is part of a directive, a given page will always provide the same content type. It is also possible to dynamically indicate the content type using the `response` object, which is discussed later. **Comments**

A JSP comment is of the form:

```
<%-- anything but a closing --%> ... --%>
```

JSP comments

The body of the JSP content is ignored completely. JSP Comments are discarded at translation time, they do not become part of the Servlet that is used to generate the response. Comments are useful for documentation but also to comment out some portions of a JSP page. JSP comments do not nest.

In order to generate comments that appear in the response to the requesting client, the HTML and XML comment syntax is used, as follows:

```
<!-- comments ... -->
```

HTML comments

Java code in the JSP file

Java code in the JSP is marked by the special characters <% and %> . To insert the value of a variable or an expression in the output it is marked with ><%= expr &> .

Be careful not to depend on the ability of JSP to include Java code too much. JSP is best used to present a `view` of data that has already been prepared in Servlets or other code, as discussed in Architecture.

Now that JSTL and JSP EL exist, they are preferred over the insertion of Java code directly.

```
<%@page session="false" contentType="text/html" import="java.util.*, example.*%>

<head>
<title>A simple thing</title>
</head>
<body>
<!-- this comment gets all the way to the browser -->
<%-- this comment gets discarded when the JSP is translated into a Servlet --%>
\textbf{\ensuremath{<}\% int x = 5; // java-style comments valid here \%\ensuremath{>}}
The value of x is \textbf{\ensuremath{<}\%= x \%\ensuremath{>}}.
The value of x + 2 is \textbf{\ensuremath{<}\%= x + 2 \%\ensuremath{>}}.
Is x
less than 6?
\textbf{\ensuremath{<}\% if (x \ensuremath{<} 6) \{ \%\ensuremath{>}}
<%@include file="y.jsp" %>
\textbf{\ensuremath{<}\% \} \%\ensuremath{>}}
</body>
```

x.jsp - Example java code in JSP file

```
Yes, it is true that x is less than 6,
with a value of \textbf{\ensuremath{<}\%= x \%\ensuremath{>}}.
```

y.jsp - Example java code in JSP file to be included

```
<head>
<title>A simple thing</title>
</head>
<body>
<!-- this comment gets all the way to the browser -->


The value of x is 5.
The value of x + 2 is 7.
Is x
less than 6?

Yes, it is true that x is less than 6,
with a value of 5.
</body>
```

x.jsp output - The result of a call to x.jsp

Importing Java packages

Usually when making a JSP the developer draws on classes from various packages. Unless these packages are imported, the use of such a class must include the fully qualified class name:

```
<%
java.util.Date d = new java.util.Date();
%>
```

Packages can be imported with the `import` attribute of the `page` directive. Multiple packages are seperated with a comma:

```
<%@ page import="java.util.*, example.*" %>
<%
Date d = new Date();
%>
```

Custom tag libraries

Custom tag libraries are a mechanism that allows developers to make functionality available to a JSP author through the use of XML tags.

As an example, a mail tag can be implemented and then used in the JSP in the following manner:

```
<mail:mailMessage
  from="registration@hogwarts.com"
  to="<%= userEmail %>"
  subject="Hogwarts Registration">

Thank-you for registering with the Hogwarts news service.  We will be sending
you an email with twice-weekly with updates and the latest news from Hogwarts.

</mail:mailMessage>
```

HogwartsRegistrationMail.jsp - an example usage of a taglib

Tag libraries can be very useful, however they are somewhat complicated to implement. They are probably best left to developers of libraries – the taglibs can provide some simple access to the functionality of the library.

More information on implementing custom tag libraries is in the JSP section of the Resin documentation.

Filter

From the Servlet Specification 2.4:

> A filter is a reusable piece of code that can transform the content of HTTP requests, responses, and header information. Filters do not generally create a response or respond to a request as servlets do, rather they modify or adapt the requests for a resource, and modify or adapt responses from a resource.

A Filter is a Java class that is used to `intercept` the request and the response ("request" and "response" are discussed later). A filter can intercept the request before it get's to other web components, such as a JSP or Servlet, and change what the request looks like. A filter can intercept the response that is generated by web components, such as a JSP or Servlet, and change (transform) the reponse before it is sent to the client.

More information on the usage of Filters is available in the Servlets and Filters section of the Resin documenation.

Custom java code

A developer can, and usually does, use the general facilities of Java to create custom Java classes that are used by the Servlets, JSP, Filters, and other components of the web applications. Java source files are placed in the `WEB-INF/classes/` directory (for example, `WEB-INF/classes/com/hogwarts/Util.java` , and are automatically compiled by Resin.

Java code libraries

A web application can take advantage of java code libraries. These libraries are usually packaged in a `.jar` file, and placed in the `WEB-INF/lib` directory (for example, `WEB-INF/lib/batik.jar`). The libraries in the jar file are then available to the Servlet, JSP, Filter, and other components of the web application.

Jar libraries usually contain code that is not specific to the web application, for example a library that provides a database driver, or a library that is used to generate images.

The structure of web applications

The componets of a web application are placed in appropriate files and configured using the `WEB-INF/web.xml` file.

Directory structure

FILE	DESCRIPTION
/index.jsp	A jsp file that is usually the default file that is accesed
/WEB-INF/web.xml	The file that configures the Web Application
/WEB-INF/classes	A directory containing classes specific to this application
/WEB-INF/classes/com/hogwarts/foo/Util.ja	
/WEB-INF/classes/com/hogwarts/foo/Util.c	

/WEB-INF/lib	A directory containing jar files
/WEB-INF/lib/mysql-connector-java-3.0.9-stable-bin.jar	An example usage of the lib directory, the MySQL driver

Web application directory structure

For the most part, the directory structure of a web application reflects the structure that the end-user will see from their browser.

In addition, a special directory named `WEB-INF` is recognized by Resin.

This directory contains a `WEB-INF/classes` sub-directory which is where you put any Java classes that are specifically for your application.

There is also a `WEB-INF/lib` directory which can contain .jar files for libraries that are needed by the application. These .jar files are usually third-party libraries or libraries that are reused amongst many different web applications.

Finally, the file `WEB-INF/web.xml` is used to configure your web application.

web.xml

A Web Application is configured with a deployment descriptor that collects information about the JSP pages, Servlets, security zones, and other resources used in the Web Application.

A full description of it can be found in the JSP specification.

A note about path seperators In the web.xml file (and in Java programming in general), always use "/" as the path seperator. Do not use the MS-DOS/Windows "\" path seperator.

Automatic compilation

A convenience that is provided by Resin is the automatic compilation of Java files that are changed.

You can take advantage of this by placing your java source files in the `WEB-INF/classes/...` directory of your web application.

For example, if you have a file `Util.java` , in package `com.hogwarts.foo` , place the file in `WEB-INF/classes/com/hogwarts/foo/Util.java` . Now any changes to `Util.java` will result in the automatic recompilation of the java file into a class file.

Lifecycle: ServletContexts, Sessions, Requests, Responses

ServletContexts, Sessions, Requests, and Responses have corresponding java objects that Resin creates. Each of these objects has a unique scope: they are created at a certain time and destroyed at a certain time.

The JavaServer Pages specification inherits from the Servlet specification the concepts of ServletContexts, Sessions, Requests and Responses.

These objects are created by Resin, they are always available to a Servlet, Filter, or JSP page.

ServletContext

A ServletContext provides a view of the application that the Servlet, Filter, or JSP is running in. The ServletContext models the Web Application. A ServletContext is an object that is common to all Servlets in an application.

A ServletContext is represented by a object.

Scope of the ServletContext Resin creates a ServletContext as soon as the Web Application is started, and it remains until the Web Application is closed. There is one ServletContext made for each web application, and any usage of the ServletContext will always use the same one.

Request and Response

Every time a user follows a URL that points to a Servlet or JSP within the Web Application, Resin creates a Request object to represent the current request. Resin also creates a Response object which the developer can use to set certain things that apply to the response to the users browser.

A request is represented by a object. A response is represented by a object.

Scope of the Request and Response A new Request and Response are created each time the user follows a link into the Web Application. These objects last until the response is sent back to the client.

Session

A Session is established for each user of the web application. The first time a user accesses anything within the web application, Resin recognizes this unknown user and creates a new Session to represent them. This Session object will remain the same for that user for the duration of their use of the web application.

A session is represented by a object.

Scope of the Session

The Session is created the first time a user access a component in the web application that requests the session. Resin sees the incoming request from a user and checks to see if it has a Session for that user. If it does not, it creates one. This session object will remain the same for that user.

The end of the session object is a bit complicated. Since Resin cannot tell the difference between when the user has stopped using the application and when the user is just taking a long time to do something, Resin must guess. Resin uses a time-out value to determine when it should assume that the user has gone away, it defaults to 30 minutes. What this means is that if a user has not accessed any resource in the Web application for 30 minutes the Resin will destroy the session object for that user.

A developer can also explicitly destroy the Session object, and that is discussed later.

How Resin keeps track of the Session Resin establishes a session by giving the user's browser a unique id that is returned back to Resin with each new request. This is accomplished in one of two ways: using cookies or URL rewriting.

Cookies Resin attempts to track the session of a user by sending the user's browser a cookie. This cookie contains a long string that Resin creates that becomes the id of the session.

URL rewriting Sometimes Resin cannot establish a cookie, either because the user has disabled cookies in their browser or because the browser does not support them (as is the case with some HDML and WML browsers). If the cookie cannot be established then something called `URL rewriting` is used.

with URL rewriting, Resin rewrites every URL that it submits to the user so that it contains a portion ';jsessionid= `id` ;'. Then for each incoming request the first thing it does is look for this parameter, and if it is available it knows a session has been established and it removes the jsessionid and uses it to find the users session object.

URL rewriting requires some assistance from the developer, which is discussed later.

Obtaining the ServletContext, Session, Request, and Response objects

Servlet

The Servlet implementation class usually obtains the ServletContext in the `init()` method. The request and response object are passed in the `doXXXX()` method. The request object is used to obtain the Session.

JSP

When translation of a JSP to a Servlet occurs, there are some variables that are automatically defined. These variables can be used within any Java code that is inserted in the page.

VARIABLE	CLASS	DESCRIPTION
response		Assists in sending a response to the client. Includes the ability to set the HTTP response headers.

request	Provides client request information including parameter name and values from forms and the HTTP headers that the client sends.
session	Provides a way to identify a user across more than one request and allows the developer to store information about that user (but see below).
application	Every Web Application gets a ServletContext when it is started. It contains information and attributes that apply to the whole application.
config	The ServletConfig for this JSP page
pageContext	A variable specific to JSP (not available to servlets). It contains a number of convenience functions for use in a JSP page.
out	An Object that is used to write into the output stream

Implicit Variables in JSP

Obtaining the session in JSP With JSP, the `page` directive is used to indicate that the JSP needs the session object. When the page is executed, and has indicated that the session is needed, a session will be created for the user if it has not already and the `session` variable will be made available.

```
<%@page session="true">
```

The default for `session` is `true` , which is convenient for pages that need the session, but creates an undue burden on the server if the page does not need the session. Therefore it is best to mark all JSP pages with `session="false"` unless they really do need the session.

```
<%@page session="false">
```

Attributes

Each of the objects `application` , `session` , and `request` can have `attributes` . These attributes consist of a `name` and a `value` .

Using attributes the developer can store a value for later retrieval.

The `name` is a String that uniquely identifies the attribute, and the `value` is a Java object.

The API for setting and getting attributes

The following methods are available for the `application` , `session` , and `request` objects.

METHOD	DESCRIPTION
getAttribute(String name)	Return the `Object` that is associated with the passed `name` . Usually you will need to cast the Object to whatever class you know it to be. If there is no attribute with the given name, `null` is returned.
setAttribute(String name, Object value)	Associates the given `name` with the `value` . If an attribute `name` already exists, it is replaced.
removeAttribute(String name)	Removes the attribute with the given `name` if it exists.

API for attributes

Choosing where to put attributes

As a developer you have a choice of which object to store attributes in. The object that you choose should be appropriate to the scope within which you will need to get that value out again.

Whenever possible put the attribute in the request object. If that cannot work, put it in the session object. And very rarely, put values in the application object.

The reason for this order is simple, all of the attributes in a session cannot be garbage collected until they are explicitly removed or the session expires. All of the objects in the application stay until they are explicitly removed or the web application is stopped or restarted.

Using the Response object

Encoding the URL

Probably the most important and most frequent usage of the response object is to rewrite the URL. Recall from earlier that in order to maintain the identity of a Session with the user Resin may have to add a special parameter to every URL. The developer must co-operate with this, using the response.encodeURL method.

Because of this, it is a good idea to form a String for every URL you will use in a page and store it in a variable:

```
<%
    String boatUrl = response.encodeURL("water/boat.jsp");
    String goatUrl = response.encodeURL("animals/goat.jsp");
%>

<%-- the presentation --%>

Hello, would you like some green eggs and ham?
Would you, could you, on a
<a href='<%= boatUrl %>'>boat</a>?
Would you, could you, with a
<a href='<%= goatUrl %>'>goat</a>?
```

EncodingUrls.jsp - Encoding the URL's and storing them in a String variable

This has the added benefit of placing all of your URL's in one place, allowing you to change them more easily.

Redirecting the users browser

Sometimes it is desirable to send a redirect to the client. This is a response to the client that tells it to go to some other URL. This is often used by developers so that the URL in the browser makes sense to the user. It is however, a slow

procedure because it requires a response to go all the way back to the browser, and then the browser will make a new request.

The redirect is accomplished by using the `response.sendRedirect()` method. Again, the URL must be encoded, and a special kind of encoding is needed for a redirect:

```
<%
   String redirectUrl =
     response.encodeRedirectURL("elsewhere.jsp");

   response.sendRedirect(redirectUrl);
%>
```

Redirect.jsp - an example of redirecting the client

A note on redirecting wireless devices Redirects often do not work with wireless devices like cell phones and WML clients, and the emulators of these devices. As well, the latency of a wireless connection compounds the speed problem of using redirects. Redirects should probably be avoided with wireless devices.

Setting the content type dynamically

As mentioned eariler, the <%@ page contentType="..." %> directive can be used to tell the browser the type of content that it is getting.

The page directive method does not allow you to do this dynamically. You can instead use: `response.setContentType("...")` to set the content type.

Telling the browser not to cache the page

Often it is necessary to inform the browser that a page should not be cached - the browser should ask for a new copy of the page every time. The best way to do this is by setting appropriate HTTP headers in the response to the browser.

Unfortunately, all the browsers work differently. The following seems to be a consensus on the headers to set:

```
<%
/** stop browser from caching */
response.setHeader("Cache-Control","no-cache,post-check=0,pre-check=0,no-cache");
response.setHeader("Pragma","no-cache");
response.setHeader("Expires","Thu,01Dec199416:00:00GMT");
%>
```

nocache.jsp - Stop the browser from caching the page

Telling the browser that the page is private

A little known fact is that it is necessary to inform the browser with certain pages that they are private pages, meaning they should never be seen by anyone except the current user.

This applies to any pages that the user has needed a password to get to.

```
<%
/**
 * Add an HTTP header to the response that
 * indicates to the browser and any
 * intervening cache's that this is a private
 * page, and should never be seen
 * by a different user.
 */
response.addHeader("Cache-Control","private");
%>
```

private.jsp - Telling the browser the page is private

Using the Request object

Retrieving the values set in form fields

The main use of the request object is to get the values that a user has supplied in a form submit. For example, with this HTML on page a.jsp:

```
<%
    String bUrl = response.encodeUrl("b.jsp");
%>

<%-- presentation --%>

<form method="post" action="<%= bUrl %>">
What is your favourite kind of eggs?
<input type="text" name="favegss" size="25">
<br>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</form>
```

a.jsp - a form that submits a value

The value that was supplied in "eggs" can be retrieved with:

```
<%
    String aUrl = response.encodeUrl("a.jsp");
%>

<%
    String faveggs = request.getParameter("eggs");
    if ((faveggs != null) && (faveggs.trim().length() == 0))
        faveggs = null;
%>

<%-- presentation --%>

<% if (faveggs == null) { %>
No eggs supplied!
<a href="<%= aUrl %>">Try again</a>
<% } else if (faveggs.equals("green"))  { %>
Of course we have <%= faveggs %> eggs!
It's one of our favourites too.
<% } else { %>
We do not have <%= faveggs %> eggs.
<% } %>
```

b.jsp - retrieving a value from a form submit

Using the Session object

The Session object is used primarily to store two types of information. The first is information about the user, and the second is data that is built up and used over multiple pages. A classic example of information you would store in a session is user profile information, or a shopping cart that is filled up over many pages.

It is important to try to limit the amount of information that is stored in the session object.

Explicitly causing the session to end

The session object will go away if the user is inactive for a certain time (usually 1/2 hour). You can also explicitly destroy the session, which you may want to do for example if the user chooses to logout.

This is accomplished with: `session.invalidate();`

Doing something when the Session goes away

It is possible to add a hook so that something can be done when Resin decides that a Session has expired, or the Session is explicitly destroyed.

To accomplish this, you write a class that implements the interface. Then you register this class with the session simply by putting it as one of the attributes. The Session will notify all of it's attributes that implement .

```
<%@page import="example.SessionListener" %>

<%
    // example.SessionListener implements HttpSessionBindingListener
    SessionListener d = new SessionListener();
    session.setAttribute("sessiongoesbyebye", d);
%>
```

SessionBind.jsp - An example of getting notification when a Session is destroyed

Warning! One session does not mean one browser window!

It is important to realize that the user can have multiple windows open that use the same session. If the user uses the "Open in new Window" functionality of a browser, they will have two windows open and the Web Application will see them both as belonging to the same session. This has some ramifications on how the Session can be used for shopping-cart like applications.

For example, let's assume that an application uses the Session attribute "cart" to store shopping cart information about different types of juice a user has chosen to purchase.

The user may start one juice buying transaction and get to the confirmation page, having 3 grape and 2 apple in their cart. Now, what happens if at this point they use another browser window to initiate another purchase? If the Web Application is using the Session to store shopping cart information then the second window may destroy the contents of the shopping cart, so that if the user finishes the first window they end up buying nothing!

A possible solution for this is to keep a `version counter` in your cart. Every time the cart is changed, the version counter is updated. Then, when your application shows the "confirm purchase" page (the next action will cause the purchase to occur), it sends the version counter to the "confirm purchase page". The "confirm purchase" page submits that version counter as a parameter. The application can check the submitted version counter, and the current version counter in the shopping cart - if they are different then the cart has been modified in some other way, and a message is returned that the purchase did not occur because the cart had been modified elsewhere.

Error Handling

JSP Translation Time Processing Errors

Some errors may occur at JSP translation time. This could be, for example, from an error in the syntax of some Java code in your JSP. You will usually see this error in your browser the first time you try to access the page. Resin can also put the error in log files.

From the JSP specification:

> The translation of a JSP page source into a corresponding JSP page implementation class using the Java technology by a JSP container can occur at any time between initial deployment of the JSP page into the runtime environment of a JSP container, and the receipt and processing of a client request for the target JSP page. If translation occurs prior to the JSP container receiving a client request for the target (untranslated) JSP page then error processing and notification is implementation dependent. Fatal translation failures shall result in subsequent client requests for the translation target to also be failed with the appropriate error; for HTTP protocols, error status code 500 (Server Error).

Coompilation Time Processing Errors

Some errors may occur at the time a Servlet or other source file is being compiled by Resin. This could be, for example, from an error in the syntax of the Java code. You will usually see this error in your browser the first time you try to access the page. Resin might also put the error in log files.

Client Request Time Processing Errors

Java handles runtime errors with exceptions. If an exception is not caught in your JSP or Servlet, Resin will use a special error page to send results back to the browser. Resin uses a default error page unless you explicitly provide an error page yourself.

From the JSP specification:

> During the processing of client requests, arbitrary runtime errors can occur in either the body of the JSP page implementation class or in some other code (Java or other implementation programming language) called from the body of the JSP page implementation class. Such errors are realized in the page implementation using the Java programming language exception mechanism to signal their occurrence to caller(s) of the offending behavior 1 . These exceptions may be caught and handled (as appropriate) in the body of the JSP page implementation class.
>
> However, any uncaught exceptions thrown from the body of the JSP page implementation class result in the forwarding of the client request and uncaught exception to the errorPage URL specified by the offending JSP page (or the implementation default behavior, if none is specified).
>
> The offending java.lang.Throwable describing the error that occurred is stored in the javax.ServletRequest instance for the client request using the putAttribute() method, using the name javax.servlet.jsp.jspException. Names starting with the prefixes java and javax are reserved by the different specifications of the Java platform; the javax.servlet prefix is used by the Servlet and JSP

specifications. If the errorPage attribute of a page directive names a URL that refers to another JSP, and that JSP indicates that it is an error page (by setting the page directive's isErrorPage attribute to true) then the exception implicit scripting language variable of that page is initialized to the offending Throwable reference.

Architecture

There are many ways to use the capabilities that are offered by JSP – different models apply to different applications.

Simple JSP

The simplest model is to use JSP as an add-on to existing web pages. At the top of each page is the Java code that is necessary for the display of the page, and at the bottom is the markup that is to be output, with appropriate insertion of any dynamically generated variables.

Note the seperation of the different logical parts. It is a very good idea to keep the presentation section as Java-less as possible. Setup all of the values in the logic part, and then use only simple <%= var %> and `if` statements in the presentation part.

```
<%-- urls --%>

<%
  // store your target urls in String variables
%>

<%-- incoming parameters --%>
<%
  // store form values from the request object in variables
%>

<%-- logic --%>
<%
  // we all make decisions with logic, right?
%>

<%-- presentation --%>

<markup>
  <forthebrowser>
  </forthebrowser>
</markup>
```

SimpleJSP.jsp - the logic and display code in one page

This seems like a sensible approach. However, imagine that later it is decided that a different set of HTML is needed for the output if there is no news available. As well, support of WML output is required. Now we need to put the

display code for 4 different types of output into one page. Maybe we need to go to different places as well, for example an administrator might get a different display page, or we need to check the users profile to see what is apporiate to display. This is too much complication to put all in one place.

This type of implementation, because of it's limitiations and it's inability to change without exponentially increasing the complexity, should be avoided.

Seperating the logic into another JSP

To address some of the disadvantages of the Simple JSP method, it is possible to seperate the 'logic' and the 'presentation' into two different JSP files.

In this model, instead of a link pointing directly to the page that does the display, it points to a page that handles the incoming request. This 'logic' page contains the Java code that is responsible for preparing the values that will be used for displaying. It stores these prepared values as attributes of the `request` object, and then `forwards` to the appropriate display page. The display page then pulls the information it needs out of the request object and displays it.

```
<%
/** -- incoming parameters -- */
     String strJuiceType = request.getParameter("juiceType");
     String strQuantity = request.getParameter("juiceQuantity");

/** -- logic -- */
Double cost;
String strCost;

here we might do a check to make sure mandatory fields
have values.

Here we might do a database lookup to determine the price, and
set the cost value to the price * quanitity

Now we might format the cost double into
a two-decimal number and store it in strCost

/** -- set outgoing parameters
    -- forward to display page -- */

/** note that here we put juiceType in as a request
    attribute, even though the display page could get
    it from the request parameters. This helps us keep
    the interface between the logic and the presentation
    cleaner */

request.setAttribute("cost", strCost);
request.setAttribute("juiceType", strJuiceType);
request.setAttribute("quantity", strQuantity);

if (isWML)
    pageContext.forward("SeperateJspPresentationWML.jsp");
else
    pageContext.forward("SeperateJspPresentationHTML.jsp");
%>
```

SeperateJsp.jsp - The logic JSP that prepares the data for display

```
<%-- urls --%>

<%-- prepared objects --%>

<%
String cost = (String) request.getAttribute("cost");
String juiceType = (String) request.getAttribute("juiceType");
String quantity = (String) request.getAttribute("quantity");
%>

<%-- presentation --%>

Hey, if you want <%= quantity %> bottles of
<%= juiceType => juice, it is going to cost
you $<%= cost %>
```

SeperateJspPresentationHTML.jsp - The JSP that presents the prepared information in HTML format

Here we have introduced a new thing: `pageContext.forward` .

pageContext.forward

The `pageContext.forward(String url)` forwards the processing to a different JSP page. That is, it passes control over to another page. This page is then executed, inheriting the current request and response objects. It is important that this is the last thing that a JSP page does.

Seperating the business logic into beans

Even the seperation of all logic into a different JSP can get a bit muddled. You can seperate the 'logic' into two types of logic, business logic and presentation logic.

Presentation logic is logic that is part of the presentation. Examples are form validation and formatting of numbers into strings.

Business logic is programming that involves your data. If you can separate something in your mind from the presentation, it doesn't depend on the presentation in any way, that's business logic. Examples are database lookups and shopping cart manipulation.

Your application will be cleaner and easier to manage if you put your business logic in Java classes (beans) which hide the complexities. You then use these classes in the logic.jsp.

Model 2 (preferred)

The final approach discussed here is called the 'Model 2' approach. This approach requires a level of sophistication that is usually not warranted unless you use a library that supports it, however it is the cleanest way to implement an application.

In this approach there is a special Servlet called a controller servlet. The controller maps an incoming request to a Java class that takes care of all of the 'business logic'. This Java class returns a result that indicates to the controller where it should go next. The controller then dispatches to the appropriate place.

The flow of a request moves through these 'business logic' pieces, finally ending up at a 'view'. The logic pieces are responsible for examing the submitted information (such as parameters from forms), managing logical flow, and preparing objects for the `view` . The objects for the view are stored as attributes of the request (or sometimes, the session).

The view is responsible for preparing a response for the user using the prepared objects now available as request or session objects. This is where JSP is often used, it works very well as the view component of a Model 2 architecture.

The developer tells the controller what the flow for each request is by specifying it an external (usually XML) file.

This approach is by far the cleanest approach to using Servlets and JSP. Most applications of any significant complexity use a Model 2 architecture.

Security

JSP inherits the Servlet security mechanism. Using the two fundamental concepts of `Principals` and `Roles` , JSP provides declarative and programmatic security.

In addition, there is builtin support for various HTTP methods of login that establish a Principal and it's Roles based on a user login and password.

Principals

A Principal is established in association with a Session. There is a one to one correspondence between the two. Usually the Principal is a user, a real live honest to goodness person sitting at a computer somewhere and accessing the Web Application.

The Application may require that the User go through a login process, a successful login will establish a Principal, which will have a certain set of Roles.

Roles

Servlet 2.2 Specification states:

> A role is an abstract logical grouping of users that is defined by the Application Developer or Assembler. When the application is deployed, these roles are mapped by a Deployer to security identities, such as principals or groups, in the runtime environment.

Roles are similar to groups in Unix security. As an example, you might have a "member" role for users who have a membership, and an "admin" role for administrators.

Principals can have more than one role – a user might be both a "member" and an "admin".

Declarative Security

Servlet 2.2 Specification states:

> Declarative security refers to the means of expressing an application's security structure, including roles, access control, and authentication requirements in a form external to the application. The deployment descriptor is the primary vehicle for declarative security in web applications.

Basically what this means in real terms is that you can specify certain paths in web.xml that require certain roles. For example, you can say that to access any page in the "/accountsetup/" subdirectory, the user must have a role of "admin".

Programmatic Security

Servlet 2.2 Specification states:

> Programmatic security is used by security aware applications when declarative security alone is not sufficient to express the security model of the application.

Programmatic security lets you decide in some Java code what to do, based on the role that a user has. For example, you might make a certain button appear in the button bar only for an "admin" user.

METHOD	DESCRIPTION
request.getRemoteUser() return the name of an authenticated user.	
request.isUserInRole(String role)	return true if the current user is in a given security role.
request.getUserPrincipal()	return the associated with the current user.

Programmatic Security API

When does Resin require a login?

The resources in an Application are by default open to the public, and do not require a login. As soon as the User attempts to access a resource that has been declared to require a role in the web.xml, Resin will insert in the process a login of some kind before continuing on to the resource that was originally requested.

For example, if the "/members/*" area has been declared to require a "member" role, as soon as the user tries to access anything in "/members/" Resin

will check to see if the user has logged in yet. If the user has logged in, then the roles for the user are checked to see if "member" is a role that user can take. If the user has not logged in, Resin inserts a login procedure and then checks the roles.

The login procedure

The Servlet Specification details numerous methods for getting user identification and password information from the user.

The example below shows usage of "Form Based Authentication", which allows you to specify a login page to be sent to the user when needed, and an error page that will be displayed if the user login fails.

The login page should submit a form to the url `j_security_check` with the parameters `j_username` and `j_password` set to appropriate values.

Resin recognizes the special url `j_security_check` and authenticates the username and password.

Lifecycle of a Principal

The Servlet authentication associates a `principal` with a session. The principal will only last as long as the Session; as soon as the Session is gone then the user will no longer be logged in.

As a result, invalidation of the Session will also cause a logout:

```
session.invalidate();
```

You can also cause a logout without invalidating the whole Session. Other objects in the session will remain available but the principal will be null:

```
session.logout();
```

Authenticating the User

Once the user information is collected, the password must be verified and the roles determined. Unfortunately, the Servlet specification does not indicate a standard way for this to occur.

What that means is that every Container is likely to do this differently, and you will have to refer to Container specific documentation and examples.

Example usage of Security

The Resin documentation on Security includes a comprehensive discussion, and the Basic Security Tutorial is a good starting point as well.

Chapter 9

Logging

9.1 Log

java.util.logging

Logger: Application logging

You can take advantage of the JDK's logging facility to add logging to your application. Choosing a good logging name and levels are important for troubleshooting and debugging your code. Logging to much can be almost as confusing as logging too little.

The logging name should be the full class name of the class you're instrumenting. Although other schemes are possible, the class name is more maintainable.

The logging level should be consistent across your application. For Resin, we use the following level conventions:

```
import java.util.logging.Logger;
import java.util.logging.Level;

public class Foo {
  private static final Logger log
    = Logger.getLogger(Foo.class.getName());

  ...
  void doFoo(String bar)
  {
    // check for log level if your logging call does anything more
    // than pass parameters
    if (log.isLoggable(Level.FINER))
        log.finer(this + "doFoo(" + bar + ")");

    ...

    log.info(...);

    try {
        ...
    } catch (ExpectedException ex) {
      log.log(Level.FINEST, "expected exception", ex);
    }
  }
  ...
}
```

Example: logging at finer

Log names

The JDK logging api uses a hierarchical naming scheme. Typically the name is aligned with a java class name. When you specify a name, all logging requests that use a name that starts with the name you have specified are matched. For example: <logger name="example.hogwarts" ...> matches a logging request for both "example.hogwarts.System" and "example.hogwarts.gryffindor.System"

Resin's logging is based on Resin's source class names. The following are useful logs.

NAME	MEANING
""	Debug everything
com.caucho.amber	Amber (JPA) handling
com.caucho.ejb	EJB handling
com.caucho.jsp	Debug jsp
com.caucho.java	Java compilation
com.caucho.server.port	TCP port debugging and threading
com.caucho.server.port.AcceptPool	port thread creation

com.caucho.server.http	HTTP-related debugging
com.caucho.server.webapp	web-app related debugging
com.caucho.server.cache	Cache related debugging
com.caucho.sql	Database pooling
com.caucho.transaction	Transaction handling

Resin log names

Log levels

The `level` for log tags matches the levels in the JDK .

NAME	API	SUGGESTED USE
off		turn off logging
severe	log.severe("...")	a major failure which prevents normal program execution, e.g. a web-app failing to start or a server restart
warning	log.warning("...")	a serious issue, likely causing incorrect behavior, like a 500 response code to a browser
info	log.info("...")	major lifecycle events, like a web-app starting
config	log.config("...")	detailed configuration logging
fine	log.fine("...")	debugging at an administrator level, i.e. for someone not familiar with the source code being debugged
finer	log.finer("...")	detailed debugging for a developer of the code being debugged
finest	log.finest("...")	events not normally debugged, e.g. expected exceptions logged to avoid completely swallowing, or Hessian or XML protocol parsing
all		all messages should be logged

Logging Level values

<log-handler>

child of: resin, server, host-default, host, web-app-default, web-app

Configure a log handler for the JDK java.util.logging.* API. java.util.logging has two steps: configure a set of log handlers, and configure the levels for each logger. The <log-handler> creates a destination for logs, sets a minimum logging level for the handler, and attaches the handler to a logging name.

In addition to configuring custom handlers, <log-handler> has the most common configuration build-in: logging to a rotating file. Most of the configuration attributes are used for the rotating file and are shared with the other logging configuration.

ATTRIBUTE	DESCRIPTION	DEFAULT
archive-format	the format for the archive filename when a rollover occurs, see Rollovers.	see below
class	configures a custom Handler class	
formatter	Configures a custom java.util.logging.Formatt to format the output.	
init	IoC-style initialization configuration for the formatter.	
level	The log level for the handler. Typically, the handler's level will be finer than the logger's level	info
mbean-name	an mbean name, see MBean control.	no mbean name, no mbean registration
name	A hierarchical name, typically aligned with the Java packaging names. The handler will be registered with the Logger with the matching name.	match all names
path	Output path for the log messages, see "Log Paths"	required
path-format	Selects a format for generating path names. The syntax is the same as for archive-format	optional

timestamp	a timestamp format string to use at the beginning of each log line.	"[%Y/%m/%d %H:%M:%S.%s] "
rollover-count	maximum number of rollover files before the oldest ones get overwritten. See Rollovers.	none
rollover-cron	cron-style specification on rollover times.	none
rollover-period	how often to rollover the log. Specify in days (15D), weeks (2W), months (1M), or hours (1h). See Rollovers.	none
rollover-size	maximum size of the file before a rollover occurs, in bytes (50000), kb (128kb), or megabytes (10mb). See Rollovers.	1mb
uri	configures a symbolic alias for the handler's class. The handler implementer will register the schema	

<log-handler> values

```
element log-handler {
  archive-format?
  & class?
  & filter?
  & format?
  & formatter?
  & init?
  & level?
  & mbean-name?
  & name
  & path?
  & path-format?
  & rollover-count?
  & rollover-period?
  & rollover-size?
  & timestamp?
  & uri?
  & use-parent-handlers?
}
```

The following example sends warning messages to a JMS queue.

The `uri="jms:"` is an alias for `com.caucho.log.handler.JmsHandler`. The `uri="timestamp:"` is a formatter alias for `com.caucho.log.formatter.TimestampFormatter`

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-queue name="myQueue" uri="memory:"/>

  <log-handler name="qa.test" level="warning" uri="jms:">
    <init target="${myQueue}"/>

    <formatter uri="timestamp:"/>
  </log-handler>

</web-app>
```

Example: logging to a JMS queue

The following example is a standard log handler writing to a rollover file. Because the handler's level is "all", the <logger> configuration will set the actual logging level.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-queue name="myQueue" uri="memory:"/>

  <log-handler name="" level="all"
       timestamp="[%Y/%m/%d %H:%M:%S.%s] {%{thread}} "/>

  <logger name="com.caucho" level="info"/>

</web-app>
```

Example: logging to a rollover file

<log>

child of: resin, server, host-default, host, web-app-default, web-app

Configure the amount and destination of debug logging for the JDK java.util.logging.* API.

ATTRIBUTE	DESCRIPTION	DEFAULT
name	A hierarchical name, typically aligned with the Java packaging names	match all names
level	The log level	info
path	Output path for the log messages, see "Log Paths"	required
path-format	Selects a format for generating path names. The syntax is the same as for archive-format	optional
timestamp	a timestamp format string to use at the beginning of each log line.	"[%Y/%m/%d %H:%M:%S.%s] "
format	a format string to control the output of each log message. Since Resin 3.0.5.	${log.message}
rollover-count	maximum number of rollover files before the oldest ones get overwritten. See Rollovers.	none
rollover-period	how often to rollover the log. Specify in days (15D), weeks (2W), months (1M), or hours (1h). See Rollovers.	none
rollover-size	maximum size of the file before a rollover occurs, in bytes (50000), kb (128kb), or megabytes (10mb). See Rollovers.	1mb
archive-format	the format for the archive filename when a rollover occurs, see Rollovers.	see below

mbean-name	an mbean name, see MBean control.	no mbean name, no mbean registration
handler	add a custom Handler, the name of a class that extends	
formatter	set a custom Formatter, the name of a class that extends	none, or if `format` is used.

<log> values

The default archive format is

```
\texttt{path}  + ".%Y%m%d"    if rollover-period >= 1 day.
\texttt{path}  + ".%Y%m%d.%H" if rollover-period < 1 day.
```

For example, to log everything to standard error use:

```
<resin xmlns="http://caucho.com/ns/resin">
  <log name='' level='all' path='stderr:' timestamp="[%H:%M:%S.%s]"/>
  ...
</resin>
```

Example: logging everything to System.err

A useful technique is to enable full debug logging to track down a problem:

```
<resin>
  ...
  <log name='' level='finer' path='log/debug.log'
       timestamp="[%H:%M:%S.%s]"
       rollover-period='1h' rollover-count='1'/>
  ...
</resin>
```

debug logging

More examples of debug logging are in the Troubleshooting section.
The class that corresponds to <log> is .

Log format string The `format` for log tags is used to specify a format string for each log message. `format` recognizes EL-expressions. The EL variable `log` is a object.

```
<log name='' level='all' path='stderr:' timestamp="[%H:%M:%S.%s]"
     format=" ${log.level} ${log.loggerName} ${log.message}"/>
```

log format string

ACCESSOR	VALUE
${log.level}	The level of the log record
${log.name}	The source loggers name
${log.shortName}	A shorter version of the source loggers name, "Foo" instead of "com.hogwarts.Foo"
${log.message}	The message, with no formatting or localization
${log.millis}	event time in milliseconds since 1970
${log.sourceClassName}	Get the name of the class that issued the logging request (may not be available at runtime)
${log.sourceMethodName}	Get the name of the method that issued the logging request (may not be available at runtime)
${log.threadID}	Get an `int` identifier of the thread where the logging request originated
${log.thrown}	Get any associated with the logging request

log EL variable 'log' is a LogRecord

You can also use the Environment EL variables in your format string:

```
<host ...>

  <web-app>
    <log name='' level='all' path='log/debug.log' timestamp="[%H:%M:%S.%s]"
         format=" [\textbf{\$\{app.contextPath\}}] ${log.message}"/>

    ...
  </web-app>

  ...

</host>
```

log format string using an Environment EL variable.

```
[14:55:10.189] [/foo] 'null' returning JNDI java:
       model for EnvironmentClassLoader[web-app:http://localhost:8080/foo]
[14:55:10.189] [/foo] JNDI lookup 'java:comp/env/caucho/auth'
       exception javax.naming.NameNotFoundException: java:comp/env/caucho/auth
[14:55:10.199] [/foo] Application[http://localhost:8080/foo] starting
```

The fmt.sprintf() function can space pad the values and make the results look a little nicer:

```
<log name='' level='all' path='stderr:' timestamp="[%H:%M:%S.%s]"
     format=" ${fmt.sprintf('%-7s %45s %s',log.level,log.loggerName,log.message)}"/>
```

fmt.sprintf() in log format string

```
[14:28:08.137] INFO com.caucho.vfs.QJniServerSocket Loaded Socket JNI library.
[14:28:08.137] INFO com.caucho.server.port.Port http listening to *:8080
[14:28:08.137] INFO com.caucho.server.resin.ServletServer ServletServer[] starting
[14:28:08.307] INFO com.caucho.server.port.Port hmux listening to localhost:6802
[14:28:08.437] INFO com.caucho.server.host.Host Host[] starting
```

fmt.sprintf() and fmt.timestamp() can be used to produce CSV files:

```
<log name='' level='all' path='log/debug.csv' timestamp=""
     format="${fmt.sprintf('%vs,%d,%d,%vs,%vs',fmt.timestamp('%Y-%m-%d %H:%M:%S.%s'),
               log.threadID, log.level.intLevel(), log.loggerName, log.message)}"/>
```

CSV log files

```
"2003-11-17 14:46:14.529",10,800,"com.caucho.vfs.QJniServerSocket",
            "Loaded Socket JNI library."
"2003-11-17 14:46:14.549",10,800,"com.caucho.server.port.Port",
            "http listening to *:8080"
"2003-11-17 14:46:14.549",10,800,"com.caucho.server.resin.ServletServer",
            "ServletServer[] starting"
"2003-11-17 14:46:14.719",10,800,"com.caucho.server.port.Port",
            "hmux listening to localhost:6802"
"2003-11-17 14:46:14.850",10,800,"com.caucho.server.host.Host",
            "Host[] starting"
"2003-11-17 14:46:15.100",10,800,"com.caucho.server.webapp.Application",
            "Application[http://localhost:8080/freelistbm] starting"
```

Log Handlers

Resin provides a number of predefined custom log handlers for common logging patterns, including sending messages to JMS, HMTP, and the syslog service. Creating your own custom handler is also straightforward.

BamHandler, uri='bam:' (3.2.0) The BAM handler publishes the log message to a BAM agent. The agent can be a custom HMTP service to process log messages. The `BamHandler` needs a JID (Jabber id) as the address of the target service.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <log-handler name="com.foo" level="warning" uri="bam:">
    <init>
       <to>test@localhost</to>
   </init>
  </log-handler>

</web-app>
```

Example: BAM handler configuration

EventHandler, uri='event:' The event handler publishes a `LogEvent` to the WebBeans event system. Any WebBeans component with an `@Observes` method for `LogEvent` will receive the notifications. The log handler classname is `com.caucho.log.handler.EventHandler` and the shortcut is uri="event:".

```
<web-app xmlns="http://caucho.com/ns/resin">

  <log-handler name="com.foo" level="warning" uri="event:"/>

</web-app>
```

Example: event handler configuration

JmsHandler, uri='jms:' The JMS handler publishes the log message to a JMS queue.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-queue name="myQueue" uri="memory:"/>

  <log-handler name="com.foo" level="warning" uri="jms:">
    <init>
       <target>${myQueue}</target>
   </init>
  </log-handler>

</web-app>
```

Example: JMS handler configuration

MailHandler, uri='mail:' (3.2.0) The Mail handler sends log messages to an email address. To keep the number of mails down, the handler will concatenate messages and only send them after a period of time.

ATTRIBUTE	DESCRIPTION	DEFAULT
to	mail address	required
delay-time	time to wait before sending first mail	1m
mail-interval-min	minimum time between mail messages	1h
properties	javamail properties in property file format	

mail: attributes

```
<web-app xmlns="http://caucho.com/ns/resin">

  <log-handler name="" level="warning" uri="mail:">
    <init>
       <to>admin@foo.com</to>
       <properties>
         mail.smtp.host=127.0.0.1
         mail.smtp.port=25
       </properties>
   </init>
  </log-handler>

</web-app>
```

Example: Mail handler configuration

SyslogHandler, uri='syslog:' On Unix systems, the SyslogHandler lets you log messages to syslog.

```
<resin xmlns="http://caucho.com/ns/resin">

<log-handler name="" level="warning" uri="syslog:">
    <init>
      <facility>daemon</facility>
      <severity>notice</severity>
    </init>
</log-handler>
```

Example: syslog configuration

The possible values for `facility` are user, mail, daemon, auth, lpr, news, uucp, cron, authpriv, ftp, local0, local1, local2, local3, local4, local5, local6, local7. The default is *daemon*.

The possible values for `severity` are emerg, alert, crit, err, warning, notice, info, debug. The default is *info*.

See also ‘ `man 3 syslog` ’ and ‘ `man syslog.conf` ’.

custom handler

```
<web-app xmlns="http://caucho.com/ns/resin">

  <log-handler name="" level="warning" class="demo.MyHandler"/>

</web-app>
```

Example: custom formatter configuration

```
package demo;

import java.util.logging.*;

public class MyHandler extends Handler
{
  @Override
  public void publish(LogRecord record)
  {
    System.out.println(getFormatter().format(record));
  }

  @Override
  public void flush();
  {
  }

  @Override
  public void close();
  {
  }
}
```

Example: MyHandler.java

Log Formatting

Sites may wish to change the formatting of log messages to gather information more appropriate for the site. The formatter can be custom-configured just like the handlers.

custom handler

```
<web-app xmlns="http://caucho.com/ns/resin">

  <log-handler name="com.foo" level="warning" path="WEB-INF/log.log">
    <formatter class="qa.MyFormatter"/>
  </log-handler>

</web-app>
```

Example: custom formatter configuration

```
package demo;

import java.util.logging.*;

public class MyFormatter extends Formatter
{
  @Override
  public String format(LogRecord record)
  {
    return "[" + record.getLevel() + "] " + record.getMessage();
  }
}
```

Example: MyFormatter.java

Standard Output Redirection

stdout-log

child of: resin, server, host-default, web-app-default, web-app**default:** use the JDK's destination for System.out

Configure the destination for `System.out` .

Usage of the `stdout-log` overrides a previous usage. For example, specifying `stdout-log` as a child of a causes a redirection of `System.out` for that web application only, and will override the `System.out` location in the enclosing .**Warning:** The `path` must not be the same as the path specified on the command line with `-stdout` . If it is, there will be conflicts with which process owns the file.

archive-format	the format for the archive filename when a rollover occurs, see Rollovers.	see below
path	Output path for the stream, see "Log Paths".	required
path-format	Selects a format for generating path names. The syntax is the same as for archive-format	optional
rollover-count	maximum number of rollover files before the oldest ones get overwritten. See Rollovers.	none
rollover-period	how often to rollover the log. Specify in days (15D), weeks (2W), months (1M), or hours (1h). See Rollovers.	none
rollover-size	maximum size of the file before a rollover occurs, in bytes (50000), kb (128kb), or megabytes (10mb). See Rollovers.	1mb
timestamp	a timestamp format string to use at the beginning of each log line.	no timestamp

The default archive format is `path` + ".%Y%m%d" or `path` + ".%Y%m%d.%H" if rollover-period $<$ 1 day.

The following example configures `System.out` for a . Unless a web-app overrides with it's own `stdout-log` , all web-apps in the host will write to the same output file.

```
...
<host id='foo.com'>
  <stdout-log path='/var/log/foo/stdout.log'
              rollover-period='1W'/>
  ...
</host>
...
```

stderr-log

child of: resin, server, host-default, web-app-default, web-app**default:** use the JDK's destination for System.err

Configure the destination for `System.err` .

Usage of the `stderr-log` overrides a previous usage. For example, specifying `stderr-log` as a child of a causes a redirection of `System.err` for that web application only, and will override the `System.err` location in the enclosing .**Warning:** The `path` must not be the same as the path specified on the command line with `-stderr` . If it is, there will be conflicts with which process owns the file.

path	Output path for the stream, see "Log Paths".	required
path-format	Selects a format for generating path names. The syntax is the same as for archive-format	optional
timestamp	a timestamp format string to use at the beginning of each log line.	no timestamp
rollover-count	maximum number of rollover files before the oldest ones get overwritten. See Rollovers.	none
rollover-period	how often to rollover the log. Specify in days (15D), weeks (2W), months (1M), or hours (1h). See Rollovers.	none
rollover-size	maximum size of the file before a rollover occurs, in bytes (50000), kb (128kb), or megabytes (10mb). See Rollovers.	1mb
archive-format	the format for the archive filename when a rollover occurs, see Rollovers.	see below

The default archive format is `path` + ".%Y%m%d" or `path` + ".%Y%m%d.%H" if rollover-period $<$ 1 day.

The following example configures `System.err` for a . Unless a web-app overrides with it's own `stderr-log` , all web-apps in the host will write to the same output file.

```
...
<host id='foo.com'>
  <stderr-log path='/var/log/foo/stderr.log'
              rollover-period='1W'/>
  ...
</host>
...
```

Access logging

child of: server, host-default, host, web-app-default, web-app

Specify the access log file.

As a child of , overrides the definition in the that the web-app is deployed in. As a child of , overrides the definition in the that the host is in.

path	Output path for the log entries, see "Log Paths".	required
path-format	Selects a format for generating path names. The syntax is the same as for archive-format	optional
format	Access log format.	see below
rollover-count	maximum number of rollover files before the oldest ones get overwritten. See Rollovers.	none
rollover-period	how often to rollover the log. Specify in days (15D), weeks (2W), months (1M), or hours (1h). See Rollovers.	none
rollover-size	maximum size of the file before a rollover occurs, in bytes (50000), kb (128kb), or megabytes (10mb). See Rollovers.	1mb
archive-format	the format for the archive filename when a rollover occurs, see Rollovers.	see below
auto-flush	true to flush the memory buffer with each request	false
resin:type	a class extending for custom logging	com.caucho.server.log.AccessLog
init	bean-style initialization for the custom class	n/a

The default archive format is `path` + ".%Y%m%d" or `path` + ".%Y%m%d.%H" if rollover-period < 1 day.

```
...
<host id=''>
  <access-log path='log/access.log'>
    <rollover-period>2W</rollover-period>
  </access-log>
  ...
</host>
...
```

The access log formatting variables follow the Apache variables:

%b	result content length
%D	time taken to complete the request in microseconds (since 3.0.16)
%h	remote IP addr
%{ xxx }i	request header xxx
%{ xxx }o	response header xxx
%{ xxx }c	cookie value xxx
%n	request attribute
%r	request URL
%s	status code
%{ xxx }t	request date with optional time format string.
%T	time taken to complete the request in seconds
%u	remote user
%U	request URI

The default format is:

```
"%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\""
```

`resin:type` allows for custom logging. Applications can extend a custom class from . Bean-style initialization can be used to set bean parameters in the custom class.

```
...
<host id='foo.com'>
  <access-log resin:type='test.MyLog'>
              path='$server-root/foo/error.log'
              rollover-period='1W'>
    <init>
      <foo>bar</foo>
    </init>
  </access-log>
  ...
</host>
...
```

Log Paths

`path` is used to configure a destination for the messages. Typically, `access-log` , `stdout-log` , and `stderr-log` are configured to go to files, and `log` is configured to go to a file or to `stderr` or `stdout` so that they show up on the console screen.

PATH	RESULT
`filesystem path`	output log entries to a file
stdout:	output log entries to stdout
stderr:	output log entries to stderr

```
  <log name="" level="all" path="stdout:"/>
```

Log messages to stdout

You can use the Environment EL variables as part of your filesystem path:

```
  <log name="" level="all"
       path="log/debug-${server.id}.log"
       rollover-period="1h" rollover-count="1"/>
```

Filesystem path using Environment EL variables

Rollovers

Log rollovers are a way to stop your log files from getting too large. When a rollover is triggered, the existing log file is renamed and a new file is started.

Size based rollover

A size based rollover is triggered when the size of the file reaches a certain amount. The default Resin behaviour for log's is to rollover when the file size reaches 1mb.

`rollover-size` is used to specify the maximum size, and can be in bytes (50000), kilobytes (128kb), or megabytes (10mb). A value of `-1` disables size based rollovers.

Time based rollover

A time based rollover is triggered when a certain period of time has passed since the last rollover. The default Resin behaviour is to perform no time based rollover, unless rollover-size has been disabled with a value of -1 in which case the default time period is 1 month.

`rollover-period` is used to specify the time period, and can be in days (15D), weeks (2W), months (1M), or hours (1h).

Archive files

When a rollover is triggered, the log file is renamed (archived) and a new log file is started.

`archive-format` is used to specify the name of the archive file. It can contain regular characters, EL Environment variables, and % codes that capture the current date and time. The % codes are the same as the ones used for `timestamp` (see Timestamp format string).

The default behaviour depends on the value of rollover-period. If rollover-period is greater than one day, or is not being used because rollover-size has been specified, the archive filename is the original path with `.%Y%m%d` appended. If rollover-period is less than one day, the archive filename is the original path with `.%Y%m%d.%H` appended.

Disabling rollovers

To completely disable rollovers, set the `rollover-size` to such a high number that it will never occur:

```
<stdout-log path="log//stdout.log" rollover-size="1024mb"/>
```

disable log rollovers

Compression

Rollover log files can be compressed with gzip or zip. The extension of the archive-format determines the compression.

```
<log name="" level="warning" path='log/error.log'
     archive-format="%Y-%m-%d.error.log.gz"
     rollover-period="1D"/>

<access-log path="log/access.log"
            archive-format="access-%Y%m%d.log.gz"
            rollover-period="1D"/>
```

Timestamp format string

The `timestamp` for log tags is a format string which can contain percent codes which are substituted with time and date values.

CODE	MEANING
%a	day of week (short)
%A	day of week (verbose)
%b	day of month (short)
%B	day of month (verbose)
%c	Java locale date
%d	day of month (two-digit)
%H	24-hour (two-digit)
%I	12-hour (two-digit)
%j	day of year (three-digit)
%m	month (two-digit)
%M	minutes
%p	am/pm
%S	seconds
%s	milliseconds
%W	week in year (three-digit)
%w	day of week (one-digit)
%y	year (two-digit)
%Y	year (four-digit)
%Z	time zone (name)
%z	time zone (+/-0800)

```
<resin xmlns="http://caucho.com/ns/resin">

  <log-handler name='' path='stderr:' timestamp="[%H:%M:%S.%s]"/>

  ...
</resin>
```

Example: typical timestamp for the log tag

```
[22:50:11.648] WebApp[/doc] starting
[22:50:11.698] http listening to *:8080
[22:50:11.828] hmux listening to *:6800
```

Chapter 10

Administration

10.1 Resin Administration

/resin-admin web-app

The /resin-admin web-app provides an administration overview of a Resin server. Resin-Pro users can obtain information across the entire cluster, profile a running Resin instance, and obtain thread dumps and heap dumps.

Configuring /resin-admin

Since /resin-admin is just a web-app implemented with Quercus/PHP, enabling it is just adding an appropriate <web-app> tag.

For security, you will also need to add a <user> to the <management> section of the resin.xml. The password will be a MD5 hash. By default, the /resin-admin web-app provides a form for generating the hash codes. You will need to copy the generated password into the resin.xml. This guarantees that you have access to the resin.xml itself to add any users. In other words, the configuration is very cautious about security issues to enable the administration.

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">

  <management>
     <user name="harry" password="MD5HASH=="/>
  </management>

<cluster id="">
<host id="">

  <web-app id="/resin-admin" root-directory="${resin.home}/php/admin">
     <prologue>
        <resin:set var="resin_admin_external" value="false"/>
        <resin:set var="resin_admin_insecure" value="true"/>
     </prologue>
  </web-app>

</host>
</cluster>
</resin>
```

resin.xml /resin-admin configuration

/resin-admin summary tab

The summary tab provides a basic overview of the Resin instance. Resin-Pro users can see the summary page of each server in a cluster.

The overview section shows basic configuration information like the server-id, Resin version, Resin home, memory information, and how long the instance has been up. It's useful as a basic check to verify the configuration and see if the server is having any problems.

Thread pool The thread pool give the current state of Resin's threads.

TCP ports The TCP ports gives information about the HTTP, HTTPS, and cluster ports, showing how many threads are active and also how many connections are in various keepalive states.

Server Connectors - Load Balancing The Server Connectors section is the main section for load balancing. It will give an overview of any failures in connecting to the backend servers, showing the latency and load.

Connection pools - Database pooling The connection pool section shows the state and history of the database pools.

WebApps The WebApps shows the current state of the active web-apps for each virtual host. In particular, it will show the time and number of any 500 errors, letting you track down errors in the log files.

/resin-admin config tab

The config tag summarizes Resin's internal configuration. This tab can be useful to double-check that the values in the resin.xml and web.xml match the expected values.

/resin-admin threads tab

The threads tab is a critical debugging tab. It shows the state and stack trace of every thread in the JVM, grouped by functionality. If the server ever freezes or moves slowly, use the thread tab as your first resource for figuring out what's going on in the system.

All Resin users should familiarize themselves with the thread dump for their application. It's very important to understand the normal, baseline status, so if something does go wrong, you'll know what looks different.

In particular, any freeze or slowness of Resin should immediately suggest looking at the thread page.

/resin-admin cpu profile tab

The cpu profile tab lets you gather basic profiling information on a running Resin server. Because the overhead is low, it's even possible to run a profile on a deployment server, which will give much better information about the performance of your system than any artificial benchmarks.

With Resin's integrated profiling, there's no excuse to skip the profiling step for your application.

/resin-admin heap dump tab

The heap dump tab lets you gather a heap memory information at any time, giving you critical information at times when you may not have a dedicated profiler available.

Admin topics

Interpreting the proxy cache hit ratio

The proxy cache is Resin's internal proxy cache (in Resin Pro). The hit ratio marks what percentage of requests are served out of the cache, i.e. quickly, and which percentage are taking the full time.

The proxy cache hit ratio is useful for seeing if you can improve your application's performance with better caching. For example, if you had a news site like www.cnn.com, you should have a high hit rate to make sure you're not overtaxing the database.

If you have a low value, you might want to look at your heavily used pages to see if you can cache more.

Resin's JMX Interfaces

See com.caucho.management.server JavaDoc.

- BlockManagerMXBean - performance data for the proxy cache, clustered sessions, and JMS queues.
- ConnectionPoolMXBean - JDBC database pools.
- HostPoolMXBean - virtual host management.
- JmsQueueMXBean - jms queue management.
- JmsTopicMXBean - jms topic management.
- LoggerMXBean - java.util.logging management.
- PortMXBean - http, https, and custom TCP ports.
- ProxyCacheMXBean - Resin's integrated proxy cache.
- ResinMXBean - Parent MXBean for Resin corresponding to the <resin> configuration tag.
- ServerConnectorMXBean - client view of a peer server in a cluster.
- ServerMXBean - information about the JVM server instance.
- SessionManagerMXBean - information about a web-app's session manager.
- ThreadPoolMXBean - information about Resin's internal thread pool.
- WebAppMXBean - information about a Resin web-app.

JMX Instrumenting Beans

Resin's IoC container can register the application's services with JMX automatically. The registered beans will then be available in a JMX application like jconsole or through PHP or Java.

1. For a class `MyFoo` , create an interface `MyFooXMBean` with the management interface.
2. Class `MyFoo` needs to implement the `MyFooMBean` interface.
3. When Resin handles the <bean> tag, it will register `MyFoo` with the JMX server.

Instrumenting a bean

Resin will automatically register any servlet which implements an MBean interface. By default, the JMX name will be:

```
resin:name= \texttt{name} ,type= \texttt{type} ,Host= \texttt{name} ,WebApp= \texttt{name}
```

ATTRIBUTE	VALUE
type	The FooMBean name minus the MBean, e.g. "Foo"
name	the bean's name value
Host	the virtual host name
WebApp	the web-app's context path

ObjectName attributes

The domain is `web-app` , the type property is calculated from the MBean class name and the name property is the value of <name>.

JMX clients will use the name to manage the bean. For example, a client might use the pattern `web-app:type=Foo,*` to retrieve the bean.

```
package test;

public interface MyServiceMBean {
  public int getCount();
}
```

MyServiceMBean.java

```
package test;

import java.io.*;
import javax.servlet.*;

public class MyService implements MyServiceMBean
{
  private int _count;

  public int getCount()
  {
    return _count;
  }

  public void doStuff()
  {
    _count++;
  }
}
```

MyServlet.java

PHP: Displaying and Managing Resources

The easiest way to display and manage JMX is with PHP. The /resin-admin web-app provides several examples of getting JMX data.

```
<php?

$mbean_server = new MBeanServer();

$service = $mbean_server->query("resin:*,type=MyService");

echo "Service.count: " . $service[0]->Count . "\n";

?>
```

PHP: Getting the Count attribute

JMX Console

JDK 5.0 includes a JMX implementation that is used to provide local and remote administration of a Resin server. The JVM will expose JMX if it's started with appropriate `-D` system properties. For example, `-Dcom.sun.management.jmxremote` will expose JMX to the local machine.

To configure the JVM arguments for Resin, you'll add a <jvm-arg> to the resin.xml. When Resin's Watchdog process starts Resin, it will pass along the configured arguments, enabling JMX administration.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

  <server-default>
    <jvm-arg>-Dcom.sun.management.jmxremote</jvm-arg>
  </server-default>

  ...

</cluster>
</resin>
```

Start Resin and allow local JMX administration

```
win> jconsole.exe
unix> jconsole

\textit{Choose Resin's JVM from the "Local" list.}
```

Start jconsole

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

  <server-default>
    <jvm-arg>-Dcom.sun.management.jmxremote.port=9999</jvm-arg>
  </server-default>

  ...

</cluster>
</resin>
```

Start Resin and allow remote JMX administration

Without some configuration effort, the previous command will not work. Password configuration and SSL configuration is required by the JDK implementation of remote JMX. Detailed instructions are included in the JDK documentation.

The following is useful for testing, but should be done with caution as the port is not protected by password or by SSL, and if not protected by a firewall is accessible by anyone who can guess the port number.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

  <server-default>
    <jvm-arg>-Dcom.sun.management.jmxremote.port=9999</jvm-arg>
    <jvm-arg>-Dcom.sun.management.jmxremote.ssl=false</jvm-arg>
    <jvm-arg>-Dcom.sun.management.jmxremote.authenticate=false</jvm-arg>
  </server-default>

  ...

</cluster>
</resin>
```

Start Resin and remote JMX - disable password checking and SSL

```
win> jconsole.exe
unix> jconsole

\textit{Enter the host name and port number (9999) on the "Remote" tab}
```

Start jconsole

```
$ cd $JAVA_HOME/jre/lib/management
$ cp jmxremote.password.template jmxremote.password
$ chmod u=rw jmxremote.password
$ vi jmxremote.password

\textit{Set a password for "monitorRole" and "controlRole":}

monitorRole 12monitor
controlRole 55control
```

Setting a password for remote JMX access

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

  <server-default>
    <jvm-arg>-Dcom.sun.management.jmxremote.port=9999</jvm-arg>
    <jvm-arg>-Dcom.sun.management.jmxremote.ssl=false</jvm-arg>
  </server-default>

  ...

</cluster>
</resin>
```

Start Resin and remote JMX - disable SSL

```
win> jconsole.exe
unix> jconsole
```

Start jconsole

Enter the host name and port number (9999) on the "Remote" tabEnter the username and password on the "Remote" tab

stat-service

Resin 3.1.6 adds a new <stat-service> to the <management tag. The <stat-service periodically checks the status of Resin and the JVM and can act if certain thresholds are exceeded. The default check rate is every 60s.

The primary statistic that <stat-service> observes is the CPU load of the server. You can set thresholds so Resin will log a thread dump if the CPU gets high, e.g. to find and debug a runaway thread. If necessary, you can also have Resin restart if the CPU load gets too high. The configuration is documented in <management> in the resin-tags section.

```
<resin xmlns="http://caucho.com/ns/resin">
  <management>
    <stat-service>
      <cpu-load-exit-threshold>10.0</cpu-load-exit-threshold>
      <cpu-load-log-info-threshold>1.0</cpu-load-log-info-threshold>
      <cpu-load-log-warning-threshold>2.0</cpu-load-log-warning-threshold>
      <cpu-load-thread-dump-threshold>2.0</cpu-load-thread-dump-threshold>
      <sample-period>15s</sample-period>
      <thread-dump-interval>10m</thread-dump-interval>
    </stat-service>
  </management>

</resin>
```

Example: CPU thresholds

SNMP

Since 3.1.5, Resin has built-in support for SNMP (Simple Network Management Protocol). This allows Resin to be managed just like any network device (e.g. routers) from an SNMP manager application.

Enabling SNMP support in Resin

To enable Resin's SNMP service, you'll need to add an SNMP protocol tag to your resin.xml under the <server> tag:

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="">
    <server-default>

      <protocol class="com.caucho.server.snmp.SnmpProtocol" port="161"/>

    <server-default/>

    ...

  <cluster/>
<resin/>
```

resin.xml

This opens up a port that listens for SNMP requests and responds to them with an SNMP response. Currently, Resin can only respond to TCP SNMP get-pdu requests.

By default, the SNMP community string is "public". It can be changed with:

```
<protocol class="com.caucho.server.snmp.SnmpProtocol" port="161">
  <init community="\textit{insert\_password\_here}"/>
<protocol/>
```

MIB Variables

Internally, Resin stores a mapping from SNMP MIB variables to MBean attributes. Requests for a specific MIB variable are simply retrievals for the corresponding MBean attribute. The available MIB mappings are hard-coded in Resin and they are:

SNMP OBJECT ID	SNMP TYPE	MBEAN	MBEAN ATTRIBUTE
1.3.6.1.2.1.1.1	Octet String	resin:type=Resin	Version
1.3.6.1.2.1.1.3	Time Ticks	java.lang:type=Ru	UpTime
1.3.6.1.2.1.1.5	Octet String	resin:type=Host,n	URL
1.3.6.1.4.1.30350.1	Gauge	resin:type=Server	KeepaliveCountTotal
1.3.6.1.4.1.30350.1	Gauge	resin:type=Server	RequestCountTotal
1.3.6.1.4.1.30350.1	Gauge	resin:type=Server	RuntimeMemory
1.3.6.1.4.1.30350.1	Gauge	resin:type=Server	RuntimeMemoryFree
1.3.6.1.4.1.30350.1	Gauge	resin:type=Server	ThreadActiveCount
1.3.6.1.4.1.30350.1	Gauge	resin:type=Server	ThreadKeepaliveCount
1.3.6.1.4.1.30350.2	Gauge	resin:type=Thread	ThreadActiveCount
1.3.6.1.4.1.30350.2	Gauge	resin:type=Thread	ThreadCount
1.3.6.1.4.1.30350.2	Gauge	resin:type=Thread	ThreadIdleCount
1.3.6.1.4.1.30350.2	Gauge	resin:type=Thread	ThreadIdleMax
1.3.6.1.4.1.30350.2	Gauge	resin:type=Thread	ThreadIdleMin
1.3.6.1.4.1.30350.2	Gauge	resin:type=Thread	ThreadMax
1.3.6.1.4.1.30350.3	Gauge	resin:type=ProxyC	HitCountTotal
1.3.6.1.4.1.30350.3	Gauge	resin:type=ProxyC	MissCountTotal

SNMP to MBean mappings

Defining your own SNMP to MBean mappings

To define your own MIB variables, you'll need to extend the `com.caucho.server.snmp.SnmpProtocol` class and then use that class name in the <protocol> tag:

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="">
    <server-default>

      <protocol class="example.MySnmpProtocol" port="161"/>

    <server-default/>

    ...

  <cluster/>
<resin/>
```

resin.xml

```
package example;

import com.caucho.server.snmp.*;
import com.caucho.server.snmp.types.*;

public class MySnmpProtocol extends SnmpProtocol
{
  public MySnmpProtocol()
  {
    super();

    addOid(new Oid("1.2.3.4.5",
                   "\textit{my\_mbean\_object\_name}",
                   "\textit{my\_mbean\_attribute}",
                   SnmpValue.OCTET_STRING));
  }
}
```

example.MySnmpProtocol

`"1.2.3.4.5"` is the SNMP ID you choose to give to your mapping. It should be in dot notation. `SnmpValue.OCTET_STRING` is the type Resin should return for that attribute. An abbreviated list of the available types are:

SNMP TYPES	**DESCRIPTION**
SnmpValue.OCTET_STRING	8-bit String
SnmpValue.INTEGER	signed 32-bit integer
SnmpValue.COUNTER	unsigned 32-bit integer that only increases, wraps around when overflows
SnmpValue.GAUGE	unsigned 32-bit integer that may increase and decrease

SnmpValue.TIME_TICKS	unsigned 32-bit integer representing time measured in hundredths of a second
SnmpValue.IP_ADDRESS	IP address

For a more complete list, see com.caucho.server.snmp JavaDoc.

Chapter 11

Deployment

11.1 Packaging/Deployment

See Also

- web-app tags defines all of the available tags for web-app configuration.
- host tags defines all of the available tags for host configuration.

Custom web-app with .war file

In this scenario, you want to configure a web-app with a specific root-directory and specify the location of the .war file. As usual, when Resin sees any changes in the .war file, it will expand the new data into the root-directory and restart the web-app. This capability, gives sites more flexibility where their directories and archive files should be placed, beyond the standard webapps directory.

The optional `archive-path` argument of the <web-app> will point to the .war file to be expanded.

ATTRIBUTE	DESCRIPTION	DEFAULT
archive-path	path to the .war file which contains the web-app's contents	
dependency-check-interval	how often Resin should check for changes in the web-app for a redeployment	2s
id	unique identifier for the web-app and the default context-path value	
redeploy-check-interval	how often Resin should check the .war file for changes	60s

redeploy-mode	how Resin should handle redeployment: automatic, lazy, or manual	automatic
root-directory	path to the expanded web-app directory	`id` as a sub-directory of the virtual-hosts's root

web-app deployment options

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">
  <host id="">

    \textbf{\ensuremath{<}web-app id="/foo" root-directory="/var/www/foo"
             archive-path="/usr/local/stage/foo.war"/\ensuremath{>}}

  </host>
</cluster>
</resin>
```

Example: resin.xml for custom web-app

web-app versioning

Resin can deploy multiple versions of a web-app simultaneously, simplifying any application upgrades. The old version of the web-app will continue to receive old sessions, while the new version will get the new requests. So any user will see a consistent version as the web site update occurs with no downtime required.

The versioning requires <web-app-deploy>, i.e. it works with the webapps directory. The versioning is numerically-based, allowing dotted notation, to determine the most recent version. A simple deployment process might use `foo-101` to upgrade from `foo-100` . A more complicated one might use `foo-10.3.14` to upgrade from `foo-10.3.13` .

The `versioning` attribute of the <web-app-deploy> enables versioning:

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">
<host id="">

  \textbf{\ensuremath{<}web-app-deploy path="webapps" versioning="true"/\ensuremath{>}}

</host>
</cluster>
</resin>
```

Example: resin.xml for webapps versioning

Deploying to a live server without interruption

It may be possible to deploy a web application to a live server without interruption to service if certain conditions are met.

1. The session objects for your users are being persisted.
2. The usage of session scoped objects between the old version and the new is compatible.
3. The usage of application scoped objects between the old version and the new is compatible.
4. Database schema changes are not required.

Resin allows you to have a backup instance running. The idea is that this backup instance of Resin takes over if your primary Resin instance goes down.

If you are using a load balancer to distribute your load to multiple primary servers, each primary server has a backup server.

You can use this feature to deploy to a live server without interruption of service.

1. shutdown primary server(s) (backup server(s) takes over)
2. deploy new war to primary server(s)
3. start primary server(s). As soon as the primary server starts, the user will be using the new version of the application.
4. deploy new war to backup server(s)

JSP Precompilation

See JSP/Compilation.

Chapter 12

Proxy Caching

12.1 Server Caching

Overview

For many applications, enabling the proxy cache can improve your application's performance dramatically. When Quercus runs Mediawiki with caching enabled, Resin can return results as fast as static pages. Without caching, the performance is significantly slower.

CACHE PERFORMANCE	NON-CACHE PERFORMANCE
4316 requests/sec	29.80 requests/sec

Mediawiki Performance

To enable caching, your application will need to set a few HTTP headers. While a simple application can just set a timed cache with `max-age` , a more sophisticated application can generate page hash digests with `ETag` and short-circuit repeated `If-None-Match` responses.

If subsections of your pages are cacheable but the main page is not, you can cache servlet includes just like caching top-level pages. Resin's include processing will examine the headers set by your include servlet and treat them just like a top-level cache.

HTTP Caching Headers

HEADER	DESCRIPTION
Cache-Control: private	Restricts caching to the browser only, forbidding proxy-caching.

Cache-Control: max-age=*n*	Specifies a static cache time in seconds for both the browser and proxy cache.
Cache-Control: s-maxage=*n*	Specifies a static cache time in seconds for the proxy cache only.
Cache-Control: no-cache	Disables caching entirely.
ETag: *hash or identifier*	Unique identifier for the page's version. Hash-based values are better than date/versioning, especially in clustered configurations.
Last-Modified: *time of modification*	Accepted by Resin's cache, but not recommended in clustered configurations.
Vary: *header-name*	Caches the client's header, e.g. Cookie, or Accept-encoding

HTTP Server to Client Headers

HEADER	**DESCRIPTION**
If-None-Match	Specifies the ETag value for the page
If-Modified-Since	Specifies the Last-Modified value for the page

HTTP Client to Server Headers

Cache-Control: max-age

Setting the `max-age` header will cache the results for a specified time. For heavily loaded pages, even setting short expires times can significantly improve performance. Pages using sessions should set a "Vary: Cookie" header, so anonymous users will see the cached page, while logged-in users will see their private page.

```
<%@ page session="false" %>
<%! int counter; %>
<%
response.addHeader("Cache-Control", "max-age=15");
%>
Count: <%= counter++ %>
```

Example: 15s cache

`max-age` is useful for database generated pages which are continuously, but slowly updated. To cache with a fixed content, i.e. something which has a valid hash value like a file, you can use `ETag` with `If-None-Match` .

ETag and If-None-Match

The `ETag` header specifies a hash or digest code for the generated page to further improve caching. The browser or cache will send the `ETag` as a `If-None-Match` value when it checks for any page updates. If the page is the same, the application will return a 304 NOT_MODIFIED response with an empty body. Resin's FileServlet automatically provides this capability for static pages. In general, the ETag is the most effective caching technique, although it requires a bit more work than `max-age` .

To handle clustered servers in a load-balanced configuration, the calculated `ETag` should be a hash of the result value, not a timestamp or version. Since each server behind a load balancer will generate a different timestamp for the files, each server would produce a different tag, even though the generated content was identical. So either producing a hash or ensuring the `ETag` value is the same is critical.

`ETag` servlets will often also use <cache-mapping> configuration to set a `max-age` or `s-maxage` . The browser and proxy cache will cache the page without revalidation until `max-age` runs out. When the time expires, it will use `If-None-Match` to revalidate the page.

When using `ETag` , your application will need to look for the `If-None-Match` header on incoming requests. If the value is the same, your servlet can return 304 NOT-MODIFIED. If the value differs, you'll return the new content and hash.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet
{
  public void doGet(HttpServletRequest req, HttpServletResponse res)
  {
    String etag = getCurrentEtag();

    String ifNoneMatch = req.getHeader("If-None-Match");

    if (ifNoneMatch != null && ifNoneMatch.equals(etag)) {
      res.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
      return;
    }

    res.setHeader("ETag", etag);

    ... // generate response
  }
}
```

Example: ETag servlet

```
C: GET /test-servlet HTTP/1.1

S: HTTP/1.1 200 OK
S: ETag: xm8vz29I
S: Cache-Control: max-age=15s
S: ...

C: GET /test-servlet HTTP/1.1
C: If-None-Match: xm8vz29I

S: HTTP/1.1 304 Not Modified
S: Cache-Control: max-age=15s
S: ...
```

Example: HTTP headers for ETag match

```
C: GET /test-servlet HTTP/1.1
C: If-None-Match: UXi456Ww

S: HTTP/1.1 200 OK
S: ETag: xM81x+3j
S: Cache-Control: max-age=15s
S: ...
```

Example: HTTP headers for ETag mismatch

Expires

Although max-age tends to be easier and more flexible, an application can also set the `Expires` header to enable caching, when the expiration date is a specific time instead of an interval. For heavily loaded pages, even setting short expires times can significantly improve performance. Sessions should be disabled for caching.

The following example sets expiration for 15 seconds. So the counter should update slowly.

```
<%@ page session="false" %>
<%! int counter; %>
<%
long now = System.currentTimeMillis();
response.setDateHeader("Expires", now + 15000);
%>
Count: <%= counter++ %>
```

Example: expires

`Expires` is useful for database generated pages which are continuously, but slowly updated. To cache with a fixed content, i.e. something which has a valid hash value like a file, you can use `ETag` with `If-None-Match` .

If-Modified-Since

The `If-Modified-Since` headers let you cache based on an underlying change date. For example, the page may only change when an underlying source page changes. Resin lets you easily use `If-Modified` by overriding methods in HttpServlet or in a JSP page.

Because of the clustering issues mentioned in the ETag section, it's generally recommended to use ETag and If-None-Match and avoid If-Modified-Since. In a load balanced environment, each backend server would generally have a different Last-Modified value, while would effectively disable caching for a proxy cache or a browser that switched from one backend server to another.

The following page only changes when the underlying 'test.xml' page changes.

```
<%@ page session="false" %>
<%!
int counter;

public long getLastModified(HttpServletRequest req)
{
  String path = req.getRealPath("test.xml");
  return new File(path).lastModified();
}
%>
Count: <%= counter++ %>
```

`If-Modified` pages are useful in combination with the `cache-mapping` configuration.

Vary

In some cases, you'll want to have separate cached pages for the same URL depending on the capabilities of the browser. Using gzip compression is the most important example. Browsers which can understand gzip-compressed files receive the compressed page while simple browsers will see the uncompressed page. Using the "Vary" header, Resin can cache different versions of that page.

```
<%
  response.addHeader("Cache-Control", "max-age=3600");
  response.addHeader("Vary", "Accept-Encoding");
%>

Accept-Encoding: <%= request.getHeader("Accept-Encoding") %>
```

Example: vary caching for on gzip

The "Vary" header can be particularly useful for caching anonymous pages, i.e. using "Vary: Cookie". Logged-in users will get their custom pages, while anonymous users will see the cached page.

Included Pages

Resin can cache subpages even when the top page can't be cached. Sites allowing user personalization will often design pages with `jsp:include` subpages. Some subpages are user-specific and can't be cached. Others are common to everybody and can be cached.

Resin treats subpages as independent requests, so they can be cached independent of the top-level page. Try the following, use the first `expires` counter example as the included page. Create a top-level page that looks like:

```
<% if (! session.isNew()) { %>
<h1>Welcome back!</h1>
<% } %>

<jsp:include page="expires.jsp"/>
```

Example: top-level non-cached page

```
<%@ page session="false" %>
<%! int counter; %>
<%
response.setHeader("Cache-Control", "max-age=15");
%>
Count: <%= counter++ %>
```

Example: cached include page

Caching Anonymous Users

The Vary header can be used to implement anonymous user caching. If a user is not logged in, he will get a cached page. If he's logged in, he'll get his own page. This feature will not work if anonymous users are assigned cookies for tracking purposes.

To make anonymous caching work, you must set the `Vary: Cookie` If you omit the Vary header, Resin will use the `max-age` to cache the same page for every user.

```
<%@ page session="false" %>
<%! int _counter; %>
<%
response.addHeader("Cache-Control", "max-age=15");
response.addHeader("Vary", "Cookie");

String user = request.getParameter("user");
%>
User: <%= user %> <%= counter++ %>
```

Example: 'Vary: Cookie' for anonymous users

The top page must still set the `max-age` or `If-Modified` header, but Resin will take care of deciding if the page is cacheable or not. If the request has any cookies, Resin will not cache it and will not use the cached page. If it has no cookies, Resin will use the cached page.

When using Vary: Cookie, user tracking cookies will make the page uncacheable even if the page is the same for all users. Resin chooses to cache or not based on the existence of any cookies in the request, whether they're used or not.

Configuration

cache

child of: cluster

<cache> configures the proxy cache (requires Resin Professional). The proxy cache improves performance by caching the output of servlets, jsp and php pages. For database-heavy pages, this caching can improve performance and reduce database load by several orders of magnitude.

The proxy cache uses a combination of a memory cache and a disk-based cache to save large amounts of data with little overhead.

Management of the proxy cache uses the ProxyCacheMXBean.

ATTRIBUTE	DESCRIPTION	DEFAULT
path	Path to the persistent cache files.	cache/
disk-size	Maximum size of the cache saved on disk.	1024M
enable	Enables the proxy cache.	true
enable-range	Enables support for the HTTP Range header.	true
entries	Maximum number of pages stored in the cache.	8192
max-entry-size	Largest page size allowed in the cache.	1M
memory-size	Maximum heap memory used to cache blocks.	8M
rewrite-vary-as-private	Rewrite Vary headers as Cache-Control: private to avoid browser and proxy-cache bugs (particularly IE).	false

<cache> Attributes

```
element cache {
  disk-size?
  & enable?
  & enable-range?
  & entries?
  & path?
  & max-entry-size?
  & memory-size?
  & rewrite-vary-as-private?
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="web-tier">
        <cache entries="16384" disk-size="2G" memory-size="256M"/>

        <server id="a" address="192.168.0.10"/>

        <host host-name="www.foo.com">
    </cluster>
</resin>
```

Example: enabling proxy cache

rewrite-vary-as-private Because not all browsers understand the Vary header, Resin can rewrite Vary to a Cache-Control: private. This rewriting will cache the page with the Vary in Resin's proxy cache, and also cache the page in the browser. Any other proxy caches, however, will not be able to cache the page.

The underlying issue is a limitation of browsers such as IE. When IE sees a Vary header it doesn't understand, it marks the page as uncacheable. Since IE only understands "Vary: User-Agent", this would mean IE would refuse to cache gzipped pages or "Vary: Cookie" pages.

With the <rewrite-vary-as-private> tag, IE will cache the page since it's rewritten as "Cache-Control: private" with no Vary at all. Resin will continue to cache the page as normal.

cache-mapping

`cache-mapping` assigns a `max-age` and `Expires` to a cacheable page, i.e. a page with an `ETag` or `Last-Modified` setting. It does not affect `max-age` or `Expires` cached pages. The FileServlet takes advantage of `cache-mapping` because it provides the ETag servlet.

Often, you want a long Expires time for a page to a browser. For example, any gif will not change for 24 hours. That keeps browsers from asking for the same gif every five seconds; that's especially important for tiny formatting gifs. However, as soon as that page or gif changes, you want the change immediately available to any new browser or to a browser using reload.

Here's how you would set the Expires to 24 hours for a gif, based on the default FileServlet.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <cache-mapping url-pattern='*.gif'
                 expires='24h'/>
</web-app>
```

Example: caching .gif files for 24h

The `cache-mapping` automatically generates the Expires header. It only works for cacheable pages setting If-Modified or ETag. It will not affect pages explicily setting Expires or non-cacheable pages. So it's safe to create a cache-mapping for `*.jsp` even if only some are cacheable.

Debugging caching

When designing and testing your cached page, it's important to see how Resin is caching the page. To turn on logging for caching, you'll add the following to your resin.xml:

```
<resin xmlns="http://caucho.com/ns/resin">

  <logger name="com.caucho.server.cache" level="fine"/>

  ...

</resin>
```

Example: adding caching log

The output will look something like the following:

```
[10:18:11.369] caching: /images/caucho-white.jpg etag="AAAAPbkEyoA" length=6190
[10:18:11.377] caching: /images/logo.gif etag="AAAAOQ9zLeQ" length=571
[10:18:11.393] caching: /css/default.css etag="AAAANzMooDY" length=1665
[10:18:11.524] caching: /images/pixel.gif etag="AAAANpcE4pY" length=61

...

[10:18:49.303] using cache: /css/default.css
[10:18:49.346] using cache: /images/pixel.gif
[10:18:49.348] using cache: /images/caucho-white.jpg
[10:18:49.362] using cache: /images/logo.gif
```

Administration

/resin-admin

block cache miss ratio The block cache miss ratio tells how often Resin needs to access the disk to read a cache entry. Most cache requests should come from memory to improve performance, but cache entries are paged out to disk when the cache gets larger. It's very important to keep the <memory-size> tag of the <cache> large enough so the block cache miss ratio is small.

proxy cache miss ratio The proxy cache miss ratio measures how often cacheable pages must go to their underlying servlet instead of being cached. The miss ratio does not measure the non-cacheable pages.

invocation miss ratio The invocation miss ratio measures how often Resin's invocation cache misses. The invocation cache is used for both cacheable and non-cacheable pages to save the servlet and filter chain construction. A miss of the invocation cache is expensive, since it will not only execute the servlet, but also force the servlet and filter chain to be rebuilt. The <entries> field of the <cache> controlls the invocation miss ratio.

BlockManagerMXBean

BlockManagerMXBean returns statistics about the block cache. Since Resin's block cache is used for the proxy cache as well as clustered sessions and JMS messages, the performance of the block cache is very important. The block cache is a memory/paging cache on top of a file-based backing store. If the block cache misses, a request needs to go to disk, otherwise it can be served directly from memory.

```
resin:type=BlockManager
```

```
public interface BlockManagerMXBean {
  public long getBlockCapacity();

  public long getHitCountTotal();
  public long getMissCountTotal();
}
```

ProxyCacheMXBean

The ProxyCacheMXBean provides statistics about the proxy cache as well as operations to clear the cache. The hit and miss counts tell how effectively the cache is improving performance.

```
resin:type=ProxyCache
```

```
public interface ProxyCacheMXBean {
  public long getHitCountTotal();
  public long getMissCountTotal();

  public CacheItem []getCacheableEntries(int max);
  public CacheItem []getUncacheableEntries(int max);

  public void clearCache();
  public void clearCacheByPattern(String hostRegexp, String urlRegexp);
  public void clearExpires();
}
```

Chapter 13

Quercus

13.1 Quercus: PHP in Java

Introduction to Quercus

Quercus is Caucho Technology's fast, open-source, 100% Java implementation of the PHP language. Quercus is a feature of Caucho Technology's Resin Application Server and is built into Resin - there is no additional download/install. Developers using Resin can launch PHP projects without having to install the standard PHP interpreter (http://www.php.net) as Quercus takes on the role of the PHP engine.

What is Quercus

Quercus implements PHP 5 and is internationalization/localization (i18n/l10n) aware. Quercus natively supports Unicode and the new Unicode syntax of the up-and-coming PHP 6. Quercus implements a growing list of PHP extensions (i.e. APC, iconv, GD, gettext, JSON, MySQL, Oracle, PDF, Postgres, etc.). Many popular PHP applications will run as well as, if not better, than the standard PHP interpreter straight out of the box.

Resin with Quercus Quercus is much more than just yet another PHP engine. Quercus is the first to tightly integrate the web server with a PHP engine. Quercus runs on top of Caucho Technology's Resin Application Server. As a result, PHP applications can automatically and immediately take advantage of Resin's advanced features like connection pooling, distributed sessions, load balancing, and proxy caching.

A New Java/PHP Architecture Quercus is pioneering a new mixed Java/PHP approach to web applications and services. On Quercus, Java and PHP is tightly integrated with each other - PHP applications can choose to use Java libraries and technologies like JMS, EJB, SOA frameworks, Hibernate, and

Spring. This revolutionary capability is made possible because 1) PHP code is interpreted/compiled into Java and 2) Quercus and its libraries are written entirely in Java. This lets PHP applications and Java libraries to talk directly with one another at the program level. To facilitate this new Java/PHP architecture, Quercus provides an API and interface to expose Java libraries to PHP.

Benefits of Quercus

Quercus and Quercus' PHP libraries are written entirely in Java, thereby taking the advantages of Java applications and infusing them into PHP. PHP applications running on Quercus are simply faster, easier to develop, more capable, more secure, and more scalable than any other PHP solution.

Quercus gives both Java and PHP developers a fast, safe, and powerful alternative to the standard PHP intepreter. Developers ambitious enough to use PHP in combination with Java will benefit the most from what Quercus has to offer.

Performance - simply faster

- Quercus outperforms straight mod_php by about 4x for MediaWiki and Drupal.
- PHP developers can use Java tools like profilers to get in-depth information about the PHP program performance.

Development - fast, safe, and easy

- PHP extensions written in Java are fast, safe, and relatively easy to develop compared to those written in C. Since Java is the library language, developers won't need to be paranoid about third-party libraries having C-memory problems or segvs and are freed to concentrate on solving the objectives at hand.

Capability - powerful Java technologies at the developer's fingertips

- Quercus has the best of both worlds: PHP and Java. PHP applications can take advantage of Java technologies like JMS, EJB, SOA frameworks, Hibernate, and Spring.

Security - no more pesky C memory bugs

- All Quercus extensions libraries are coded in Java. Therefore, developers do not have to worry about C pointer overruns and segmentation faults from PHP extensions anymore.

Scalability - Massive clusters of PHP

- Thanks to Resin, PHP applications can beautifully scale to as many servers as desired.
- PHP applications can now enjoy connection pooling, distributed sessions, fail-safe load balancing, and proxy caching. These benefits require no change in the PHP code.

Internationalization - 16-bit unicode

- Because Quercus is a Java implementation, it natively supports 16-bit unicode strings and functions. Quercus (in 3.1.0) supports the new PHP 6 internationalization syntax, and the older unicode conversion functions like `iconv` . Since 3.1.3, the new PHP6 unicode features are off by default but they can be enabled with the PHP ini `unicode.semantics=on` .

Existing PHP applications on Quercus

Killer Apps: Mediawiki, Wordpress

Caucho has designated a few applications as Quercus "killer apps". For these applications, we take extra time to test that each new application version works well with Quercus. Any issues raised with the killer apps have priority over other Quercus bugs.

- http://wiki.caucho.com/Mediawiki
- http://wiki.caucho.com/Wordpress

Other applications

- http://wiki.caucho.com
- DokuWiki
- Drupal
- Gallery2
- Mantis
- Mediawiki
- Openads
- PHP-Nuke
- phpMyAdmin
- PHProjekt
- Vanilla
- Wordpress

Configuring Quercus

php.ini

Individial PHP initialization values can be set in resin-web.xml. For example, to set the settings for sending mail:

```
<web-app xmlns="http://caucho.com/ns/resin">
  <servlet-mapping url-pattern="*.php"
                   servlet-class="com.caucho.quercus.servlet.QuercusServlet">
    <init>
      <php-ini>
        <sendmail_from>my_email_address</sendmail_from>
        <smtp_username>my_email_username</smtp_username>
        <smtp_password>my_email_password</smtp_password>
      </php-ini>
    </init>
  </servlet-mapping>
</web-app>
```

WEB-INF/resin-web.xml

A PHP style ini file can also be specified:

```
<web-app xmlns="http://caucho.com/ns/resin">
  <servlet-mapping url-pattern="*.php"
                   servlet-class="com.caucho.quercus.servlet.QuercusServlet">
    <init>
      <ini-file>WEB-INF/php.ini</ini-file>
    </init>
  </servlet-mapping>
</web-app>
```

WEB-INF/resin-web.xml

Character Encoding

Quercus 3.1.0 supports PHP6 and has full support for Unicode. But like PHP6, Quercus 3.1.3 has its Unicode support turned off by default for compatibility with legacy PHP applications. Unicode support can be enabled with the php ini `unicode.semantics` .

With unicode semantics off, Quercus will intrepret bytes in the default ISO-8859-1 encoding. Quercus will behave just as PHP5 would. With it on, PHP5 applications may break and you need to be concerned with the following three encodings options: script-encoding, unicode.output_encoding, and unicode.runtime_encoding. By default, Quercus uses UTF-8 for all three.

Script encoding indicates the encoding of PHP script source files. If the source code for an application is not encoded in UTF-8, Quercus may give invalid UTF-8 conversions errors when it tries to convert bytes read to UTF-8. The solution is to tell Quercus to parse PHP scripts using the correct character

set (ISO-8859-1 for most applications). For example, to tell Quercus to use ISO-8859-1, add <`script-encoding`> to the `init` tag of QuercusServlet:

```
<web-app xmlns="http://caucho.com/ns/resin">

  <servlet-mapping url-pattern="*.php"
                   servlet-class="com.caucho.quercus.servlet.QuercusServlet">
    <init>
      \textbf{\ensuremath{<}script-encoding\ensuremath{>}ISO-8859-1\ensuremath{<}/script-encoding\ensurem
    </init>
  </servlet-mapping>

</web-app>
```

WEB-INF/resin-web.xml

If the PHP application also expects conversion from binary to string using a character encoding that is not UTF-8, then the `unicode.runtime_encoding` is used to specify the encoding. In PHP 6, there are two types of strings, Unicode and binary. A binary string is a string where the data is binary, the encoding is unknown, or the encoding is not Unicode (UTF-16). If you ever use a function that will likely return a binary string, then you probably need to set unicode.runtime_encoding. Quercus may convert your binary string to Unicode and then to your output encoding for output to the browser. If your runtime encoding is wrong, then you would see garbage in your browser.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <servlet-mapping url-pattern="*.php"
                   servlet-class="com.caucho.quercus.servlet.QuercusServlet">
    <init>
      <script-encoding>iso-8859-1</script-encoding>
      <php-ini>
        \textbf{\ensuremath{<}unicode.runtime\_encoding\ensuremath{>}iso-8859-1\ensuremath{<}/unicode.run
      </php-ini>
    </init>
  </servlet-mapping>

</web-app>
```

WEB-INF/resin-web.xml

unicode.output_encoding is the charset used to display output to the browser. You can set it in your resin-web.xml:

```
<web-app xmlns="http://caucho.com/ns/resin">
  <servlet-mapping url-pattern="*.php"
                   servlet-class="com.caucho.quercus.servlet.QuercusServlet">
    <init>
      <script-encoding>iso-8859-1</script-encoding>
      <php-ini>
        \textbf{\ensuremath{<}unicode.output\_encoding\ensuremath{>}iso-8859-1\ensuremath{<}/u
        <unicode.runtime_encoding>iso-8859-1</unicode.runtime_encoding>
      </php-ini>
    </init>
  </servlet-mapping>
</web-app>
```

WEB-INF/resin-web.xml

Compiling PHP Scripts for Increased Performance

Quercus will automatically compile PHP scripts into Java classes for better performance. This is available only in Resin Professional.

The default behaviour in Resin Professional is to execute the PHP script in interpreted mode, and to compile the script in the background. When the compiled version is ready, it is used instead of the interpreted version. To force compilation, use the <`compile`> tag within the <`init`> tag:

```
<web-app xmlns="http://caucho.com/ns/resin">
  <servlet-mapping url-pattern="*.php"
                   servlet-class="com.caucho.quercus.servlet.QuercusServlet">
    <init>
      \textbf{\ensuremath{<}compile\ensuremath{>}true\ensuremath{<}/compile\ensuremath{>}}
    </init>
  </servlet-mapping>
</web-app>
```

WEB-INF/resin-web.xml

Using Databases

JDBC drivers are required to use databases in Quercus. There are JDBC drivers for MySQL, Oracle, SQLite, and many other database engines. The desired JDBC driver should be downloaded into Resin's `${resin.root}/lib` directory. Resin will automatically load jars in the `lib` directory upon startup.

DATABASE TYPE	URL FOR DOWNLOAD
MySQL	http://dev.mysql.com/downloads/connector/j
PostgreSQL	http://jdbc.postgresql.org/download.html
Oracle	http://www.oracle.com/technology/software/tech/java/sqlj_jdb

The database support in Quercus supports robust database connection pooling since Quercus runs in Resin, a fast Java application server. All PHP database access automatically uses JDBC-pooled connections. PHP code does not need changing to take advantage of this capability.

The PHP database apis supported include PDO (portable database objects), mysql, mysql improved, postgres and oracle. Any JDBC-compliant database is available to PHP scripts using PDO.

```
<php

$db = new PDO("java:comp/env/jdbc/my-database");

...
?>
```

PDO access to JNDI-configured databases

JNDI DataSource If a database with JNDI name `jdbc/myDatabase` is defined in resin.xml, (see Database Configuration), Quercus can do a JNDI lookup for the database when database functions are called. Thus, database connection parameters like user name can be omitted within PHP scripts. This allows easier maintenance and enables Java and PHP database settings to be centrally located in resin.xml.

Scripts can use the jndi name directly:

```
<?php

  // standard PHP
  //mysql_connect($host, $username, $password, $dbname);

  // using JNDI lookup
  mysql_connect("java:comp/env/jdbc/myDatabaseName");

?>
```

You can use a JNDI `<database` configuration in the WEB-INF/resin-web.xml to override the PHP connection code. If a <database> is provided, any `mysql_connect` call will return the configured database, ignoring the parameters to the `mysql_connect` call.

```
<web-app xmlns="http://caucho.com/ns/resin">
  <database jndi-name="jdbc/mysql">
    <driver type="org.gjt.mm.mysql.Driver">
      <url>jdbc:mysql://localhost:3306/test</url>
      <user></user>
      <password></password>
    </driver>
  </database>

  <servlet-mapping url-pattern="*.php"
                   servlet-class="com.caucho.quercus.servlet.QuercusServlet">
    <init>
      \textbf{\ensuremath{<}database\ensuremath{>}java:comp/env/jdbc/myDatabaseName\ensuremath
    </init>
  </servlet-mapping>
</web-app>
```

Example: overriding database configuration in resin-web.xml

Java/PHP integration

Using Java services from PHP: Resin-IoC/WebBeans

If you're already using Resin-IoC/WebBeans to organize your application into services, your PHP script can grab a reference to the service with the registered name. Calling `java_bean` with a singleton bean's name will return the singleton, and calling `java_bean` with a component's name will return a new instance of the component. Quercus will automatically marshal PHP calls to the bean's Java methods, so your service's entire functionality will be available to the PHP script.

```
<?php

  $house_manager = java_bean("houseManager");

  $house = $house_manager->findHouse("Gryffindor");

  foreach ($house->getPrefects() as $prefect) {
    echo $prefect . "\n";
  }

?>
```

Example: using a WebBeans service

WebBeans (JSR-299) is an inversion-of-control/dependency-injection framework specification for JavaEE 6 which is designed to organize Java services, and also integrate with scripting frameworks. WebBeans integrates tightly with the newest EJB and persistence specifications, so PHP applications using the

WebBeans interface will gain the latest, cleanest integration with Java applications. Resin-IoC is Caucho's implementation of the WebBeans framework, and serves as the underlying architecture of Resin itself, as well as the Resin EJB implementation.

Working with Java classes in PHP

Instantiating objects by class name An alternative to `import` is to use `new Java(...)` with the class name and any constructor arguments.

```
<?php

  $a = new Java("java.util.Date", 123);

  echo $a->time;

?>
```

Importing classes Quercus supports the use of an import statement in PHP. `import` makes any java class available to the PHP script with it's unqualified name.

```
<?php

  import java.util.Date;

  $a = new Date(123);

  echo $a->time;
?>
```

Example: creating a Java class with import

User classes can be placed in the webapp's WEB-INF/classes directory.

```
package example;

public class MyBean
{
  int _value;

  public MyBean(int value)
  {
    _value = value;
  }

  public int getValue()
  {
    return _value;
  }

  public String makeMessage()
  {
    return "Hello, my value is " + _value;
  }
}
```

Example: WEB-INF/classes/example/MyBean.java

```
<?php

  import example.MyBean;

  $bean = new MyBean(123);

  var_dump($bean);
  var_dump($bean->value);
  var_dump($bean->makeMessage());
?>
```

mybean.php

The `import` keyword will also work on PHP classes but it has a different functionality than for Java classes. `import` will try to autoload PHP classes by including the file WEB-INF/classes/*classname*.php from the application's WEB-INF/classes directory.

Calling Java Methods

PHP syntax is used for invoking methods. PHP property syntax can be used for invoking getters and setters of Java objects.

```
<?php

  import java.util.Date;

  $a = new Date(123);

  echo $a->getTime();    # calls getTime()
  echo $a->setTime(456); # calls setTime(456)

  echo $a->time;       # calls getTime()
  $a->time = 456;      # calls setTime(456)
?>
```

Static members and methods Static methods and members are available using PHP syntax if the Java class has been imported.

```
<?php

import java.util.Calendar;

$calendar = Calendar::getInstance();

var_dump($calendar);
?>
```

An alternative to `import` is to use `java_class()` to access static members and methods.

```
<?php

  $class = java_class("java.lang.System");

  # System.in
  $in = $class->in;

  # System.currentTimeInMillis();
  $time = $class->currentTimeInMillis();

?>
```

Java method overloading Quercus allows overloaded Java methods to be called from within PHP code. The number of arguments is most important, followed by the argument types. Quercus will use the method whose arguments are the most easily marshaled (i.e. a PHP string easily goes into a Java String whereas a PHP array is a mismatch for a Java int).

Because the PHP language itself does not support overloading, the Quercus overloading of Java methods may not be exact. Therefore, it's best to keep overloading to a minimum. Overloading by the number of arguments will always work, but overloading by types is trickier.

```
import com.caucho.quercus.module.AbstractQuercusModule;

public class MyModule extends AbstractQuercusModule
{
  public static void foo(String a, boolean b)
  {
  }

  public static void foo(String a, String b)
  {
  }
}
```

Example: MyModule.java

```
<?php

  foo('abc', false);

?>
```

example.php

In the example above, the first Java method `public static void foo(String a, boolean b)` is called because it requires the least amount of type coercion.**Note:** Only Java methods with the same amount of arguments will be considered.

Modules: Adding PHP functions

The core PHP functions are implemented inside Quercus modules. Quercus modules are the Java equivalent of PHP modules.

All Quercus modules need to implement AbstractQuercusModule. Functions defined in your modules are callable from within PHP script by using just the function name. Function names need to be distinct in order to prevent name collisions, though Quercus does support function overloading (for Java functions only).

A typical Quercus module looks like:

```
package example;

import com.caucho.quercus.env.Env;
import com.caucho.quercus.module.AbstractQuercusModule;

public class HelloModule extends AbstractQuercusModule
{
  /**
   * @param env provides Quercus environment resources.
   * @param str
   */
  public void hello_test(Env env, String str)
  {
    // 'echos' the string
    env.println("hello " + str);
  }
}
```

WEB-INF/classes/example/HelloModule.java

```
<?php

  // PHP 5 is case-insensitive
  // just prints "hello me" to the browser.
  hello_test("me");

?>
```

example.php

For a tutorial on how to implement your own Quercus module, see the Quercus module tutorial.

Marshalling: PHP to Java conversions

PHP types For every PHP type, there is a Java type that is used internally to represent the corresponding PHP value. All of the Java types extend `Value` .

PHP TYPE	QUERCUS CLASS
null	NullValue
string (php5)	StringBuilderValue
string (php6, binary)	BinaryBuilderValue
string (php6, unicode)	UnicodeBuilderValue
bool	BooleanValue
int	LongValue
float	DoubleValue
array	ArrayValue

object	ObjectValue
ref/var	Var

Java method arguments In Quercus, Java methods can be called from within PHP. Java arguments for Java methods are marshaled to the correct type from the PHP parameters that were passed in.

When the Java argument type is declared to be Object, the value will be marshaled to a Java object. For example, a PHP int (LongValue) will be marshaled to an Integer. The only exceptions are PHP arrays and objects: they are passed in as-is without marshaling.

When the Java argument type is declared to be a Quercus Value, the PHP value is passed in directly without marshaling.

If the Java argument type is an object, passing in a PHP `NULL` will result in a null Java argument.

JAVA TYPE	**PHP TYPE**	**QUERCUS TYPE**
null	NULL	NullValue
boolean	bool	BooleanValue
Boolean	bool	BooleanValue
byte	int	LongValue
Byte	int	LongValue
short	int	LongValue
Short	int	LongValue
int	int	LongValue
Integer	int	LongValue
long	int	LongValue
Long	int	LongValue
float	float	DoubleValue
Float	float	DoubleValue
double	float	DoubleValue
Double	float	DoubleValue
String	string (php5)	StringBuilderValue
String	unicode (php6)	UnicodeBuilderValue
char	string (php5)	StringBuilderValue
char	unicode (php6)	UnicodeBuilderValue
Character	string (php5)	StringBuilderValue
Character	unicode (php6)	UnicodeBuilderValue
char[]	string (php5)	StringBuilderValue
char[]	unicode (php6)	UnicodeBuilderValue
byte[]	string (php5)	StringBuilderValue
byte[]	string (php6)	BinaryBuilderValue
Object[] (any other array)	array	ArrayValue

Calendar	effectively int (getTimeInMillis())		JavaValue
Date	effectively int (getTime())		JavaValue
URL	effectively (toString())	string	JavaValue
Collection	array		JavaValue
List	array		JavaValue
Map	array		JavaValue
other Java Objects	object		JavaValue
Value	any		Value

Java to PHP conversion

Java objects like Calendar and Map are placed inside JavaValues and then returned to the PHP environment. A JavaValue is a wrapper that exposes the object's Java methods to PHP. For example, if `$url` is holding a Java URL object, then we can use `$url->getHost()` to call the URL's `getHost()` method.

Some Java objects may have an effective PHP value. Take for instance, Date. A Date object is, for practical purposes, a PHP int with it's value pegged to Date.getTime().

Collection, List, and Map behave just like PHP arrays. Suppose `$map` holds a Java HashMap, then it's certainly valid to do `$map["foo"] = "bar"` . However, there are some limitations that are dependent on the underlying Java type. For example, `$list[-1] = 5` will not be possible for a Java List because List indexes start at 0.

HttpServletRequest and HttpSession

QuercusServlet automatically creates the `$request` variable for PHP scripts. It contains the object.

PHP sessions are not shared with servlet sessions. The `$request` variable can be used to obtain the servlet session if required.

```
<?php
  $session = $request->getSession(true);

  $foo = $session->getAttribute("foo");
?>
```

`$request->getSession(true) HttpSession`

Using META-INF/services to package Quercus extensions in jar files

At startup time Quercus discovers and uses files within the META-INF/services to discover modules and classes that are made available within the PHP environment.

META-INF/services/com.caucho.quercus.QuercusModule

`META-INF/services/com.caucho.quercus.QuercusModule` contains a list of classes that are modules. Each public member of a module becomes available as a top-level PHP function.

For example a line like:

```
...
com.caucho.quercus.lib.date.DateModule
...
```

META-INF/services/com.caucho.quercus.QuercusModule

references the java class:

```
package com.caucho.quercus.lib.date.DateModule

public class DateModule
  extends AbstractQuercusModule
{
  ...

  public static final int CAL_GREGORIAN = 0;

  ...

   public static int cal_days_in_month(int cal, int month, int year)
   {
     ...
   }

   ...
}
```

com/caucho/quercus/lib/date/DateModule.java

When Quercus starts, the DateModule class is automatically introspected, and a PHP script can call the function defined in the module:

```
<?php
$var = cal_days_in_month(CAL_GREGORIAN, 2, 2000)
?>
```

Calling the cal_days_in_month function

META-INF/services/com.caucho.quercus.QuercusClass

`META-INF/services/com.caucho.quercus.QuercusClass` contains a list of classes that should become available to PHP as objects.

For example a line like:

```
...
com.caucho.quercus.lib.date.DateTime
...
```

META-INF/services/com.caucho.quercus.QuercuClass

references the java class:

```
package com.caucho.quercus.lib.date;

public class DateTime
{
  public static final String ISO8601 = "Y-m-d\\TH:i:sO";

  public DateTime(String timeString)
  {
    ...
  }

  ...

  public String format(String format)
  {
    ...
  }

   ...
}
```

com/caucho/quercus/lib/date/DateTime.java

When Quercus starts, the DateTime class is automatically introspected, and a PHP script can instantiate and use the object:

```
<?php
$var = new DateTime("last week");

echo $a->format(DateTime::ISO8601);
?>
```

Using the DateTime class in PHP

PHP Module highlights

Standard modules

Quercus implements the standard PHP libraries (arrays, strings, date, regexp, etc). It also supports extension libraries like zip and zlib for compression, mcrypt for encryption, mail (implemented with JavaMail), and bcmath for large numbers.

APC (object caching)

For PHP object caching, Quercus implements the APC module. PHP applications can use the APC functions to save PHP objects without resorting to serialization and database persistence. Because Quercus runs in Resin, a Java web server, the saved objects are quickly available to any thread running PHP. In other words, unlike Apache which makes sharing across different PHP processes difficult, Quercus can just store a singleton cache of the APC-cached objects.

Because Quercus compiles PHP to Java code, PHP scripts get the opcode caching of APC for free. At this time, performance of Quercus is roughtly comparable with performance of mod_php with APC, i.e. it is significantly faster (3-5 times) than mod_php running by itself.

Image support ('gd')

Quercus provides the image module, so users can use image manipulation functions like watermarking and thumbnail generation in any PHP script on Quercus. .jpg, .png, and .gif files are currently supported. Java users will also find these libraries convenient.

PDF generation (PDFlib api)

PDF generation in Quercus follows the PDFlib API. Since the Quercus PDF implementation is a new implementation in Java, no special downloads are needed to use PDF.

AJAX (JSON)

Quercus also includes the JSON module for encoding and decoding AJAX-style requests. The JSON modules is an excellent example of the benefits of writing modules in Java. Because Java supports garbage collection and protects from pointer overruns, the JSON module implementation is straightforward and reliable, without having to worry about all the possible memory problem in a C library.

Gettext (localization)

Quercus supports the gettext API and .po and .mo files. gettext is a portable API for localization, i.e. translation of program messages. In the future, the Quercus gettext implementation will support Java message bundles so Java applications using PHP can use standard Java localization techniques.

ResinModule

jndi_lookup

Retrives an object from JNDI. jndi_lookup is useful in a SOA (Service Oriented Architecture) system to locate a Java service.

```
<?php

$conn = jndi_lookup("java:comp/env/jms/jms-connection-factory");
$queue = jndi_lookup("java:comp/env/jms/test-queue");

...
?>
```

mbean_explode

Explodes a JMX ObjectName into an array.

```
<?php

var_dump(mbean_explode("resin:type=WebApp,name=/foo,Host=bar.com"));

?>
```

mbean_explode

```
array(4) {
  [":domain:"]=>
  string(5) "resin"
  ["Host"]=>
  string(7) "bar.com"
  ["name"]=>
  string(4) "/foo"
  ["type"]=>
  string(6) "WebApp"
}
```

mbean_implode

Creates a JMX ObjectName from an array.

```
<?php

$a = array(":domain:"=>"resin", "type" => "ThreadPool");

var_dump(mbean_implode($a));

?>
```

mbean_implode

```
resin:type=ThreadPool
```

MBeanServer

An object representing a JMX MBeanServer.

```
<?php

$mbeanServer = new MBeanServer();

$threadPool = $mbeanServer->lookup("resin:type=ThreadPool");

echo "thread-max: " . $threadPool->threadMax;
```

lookup Returns a proxy to the mbean matching the given name.

```
<?php

$mbeanServer = new MBeanServer();

$threadPool = $mbeanServer->lookup("resin:type=ThreadPool");
```

query Returns mbean proxies matching the name pattern.

```
<?php

$mbeanServer = new MBeanServer();

foreach ($webApp in $mbeanServer->query("resin:type=WebApp,*")) {
  echo $webApp->name . "<br>\n";
}
```

resin_debug

Write debugging information to the log. The log is at INFO level.

```
<?php

$a = array("a", "b");

resin_debug("ARRAY: $a[0]");

?>
```

resin_thread_dump

resin_thread_dump

Produce a thread_dump to the logs. The log is at INFO level.

```
<?php

$a = array("a"=>"b");

resin_thread_dump();

?>
```

resin_thread_dump

resin_call_stack

Returns an array containing the current PHP function call stack.

```
<?php

function foo()
{
  bar();
}

function bar()
{
  var_dump(resin_call_stack());
}

foo();

?>
```

resin_call_stack

resin_var_dump

Produce a var_dump to the logs. The log is at INFO level.

```
<?php

$a = array("a"=>"b");

resin_var_dump($a);

?>
```

resin_var_dump

resin_version

Returns the version of Resin running Quercus.

```
<?php

var_dump(resin_version());

?>
```

xa_begin

Starts a distributed transaction. All database connections will automatically participate in the transaction. Returns TRUE for success, FALSE for failure.

```
<?php

xa_begin();

...

xa_commit();
?>
```

xa_commit

Commits a distributed transaction. All database connections will automatically participate in the transaction. Returns TRUE for success, FALSE for failure.

```
<?php

xa_begin();

...

xa_commit();
?>
```

xa_rollback

Rolls back a distributed transaction. All database connections will automatically participate in the transaction. Returns TRUE for success, FALSE for failure.

```
<?php

xa_begin();

...

xa_rollback();
?>
```

xa_rollback_only

Marks the current distributed transaction as rollback only. Subsequent attempts to commit the transaction will fail with a warning. Returns TRUE for success, FALSE for failure.

See Also

- Quercus home page
- Caucho Forums, including a Quercus forum
- Caucho Bug Tracker
- Caucho Mailing Lists

Chapter 14

Security

14.1 Resin Security

Quick Start

The following sample shows how to protect a section of a web-site with a password, using a login form.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <security-constraint url-pattern="/secure/*">
    <auth-constraint role-name="*"/>

    <login uri="form:login-page=/login.jsp"/>
  </security-constraint>

  <authenticator uri="properties:password-digest=none">
     <init>
       harry=quidditch,user
     </init>
  </authenticator>

</web-app>
```

WEB-INF/resin-web.xml - Simple Password Protection

- <security-constraint> protects a section of the web-app, i.e. providing an authorization context.
- <url-pattern> matches the URLs to be protected
- <auth-constraint> protects the web-app through login (as opposed to by IP address or by SSL)
- <login> specifies the login method

- <authenticator> defines the login users and passwords. The "properties:" authenticator specifies a simple .properties file user definition.

- password-digest=none disables the default MD5-digest for the passwords. Only recommended for examples.

Management

Since all Resin users will want to protect the `/resin-admin` pages with an administration password, and protect any clustered management and deployment, Resin's top-level <management> tag includes a static, XML-based authentication context. The authenticator is automatically shared for all hosts and web-apps, so simple sites can even use this authenticator configuration for their site-wide authentication.

```
<resin xmlns="http://caucho.com/ns/resin">

  <management">
     <user name="admin" password="MD5HASH=="/>

     ...
  </management>

  ...

</resin>
```

resin.xml

The password is a hash of the user name, password, and the "resin" realm. The `/resin-admin` page includes a form to easily generate the MD5 hash. You can also use the PasswordDigest class to generate the digest programmatically.

Authentication

Resin provides a basic set of authenticators covering the most common cases. Applications which need custom authenticators can easily write their own extensions, described below.

properties: - properties authentication

```
<web-app xmlns="http://caucho.com/ns/resin">

  <authenticator uri="properties:password-digest=none">
     <init>
       harry=quidditch,user,admin
       draco=mudblood,disabled,user
     </init>
  </authenticator>

</web-app>
```

WEB-INF/resin-web.xml - inline properties

```
<web-app xmlns="http://caucho.com/ns/resin">

  <authenticator uri="properties:path=WEB-INF/users.properties"/>

</web-app>
```

WEB-INF/resin-web.xml - file property

```
harry=MD5HASH==,user,admin
```

WEB-INF/users.properties

xml: - xml authentication

```
<web-app xmlns="http://caucho.com/ns/resin">

  <authenticator uri="properties:password-digest=none">
     <init>
       <user name="harry" password="quidditch"/>
     </init>
  </authenticator>

</web-app>
```

WEB-INF/resin-web.xml - inline xml

```
<web-app xmlns="http://caucho.com/ns/resin">

  <authenticator uri="properties:path=WEB-INF/users.xml"/>

</web-app>
```

WEB-INF/resin-web.xml - file xml

```
<users>
  <user name="harry password="MD5HASH==" roles="user,admin"/>
<users>
```

WEB-INF/users.xml

custom authentication

```
<web-app xmlns="http://caucho.com/ns/resin">

  <authenticator class="com.foo.MyAuthenticator">
    <init>
      <foo>bar</foo>
    </init>
  </authenticator>

</web-app>
```

WEB-INF/resin-web.xml - custom

```
package com.foo;

import com.caucho.server.security;

public class MyAuthenticator extends AbstractPasswordAuthenticator {
  private PasswordUser _user;

  public MyAuthenticator()
  {
    _user = new PasswordUser("harry", "quidditch",
                             new String[] { "user" });
  }
192
  public PasswordUser getUser(String userName)
  {
    if (userName.equals(_user.getName()))
      return _user;
    else
      return null;
  }
}
```

MyAuthenticator.java

It's also possible to register your custom authenticatr with Resin's uri-based configuration. You'll add a file in the `META-INF/services/com.caucho.config.uri` named `com.caucho.server.security.ServletAuthenticator` in the .jar file with the following contents:

```
foo.my=com.foo.MyAuthenticator
```

com.caucho.server.security.ServletAuthenticator

Quick Start

The easiest authenticator to understand is the XmlAuthenticator. It lets you put users and passwords directly in the configuration file. The following example uses "Basic" authentication for login. Basic authentication asks the browser to pop open a window prompting for a username and password. (Basic authentication is discouraged because it is not secure unless you use it with SSL, but it's the easiest example.) The only user defined here is "Harry Potter" and he has the password "quidditch". He also plays the "user" role.

```
<web-app xmlns="http://caucho.com/ns/resin">

  ...

  <authenticator type="com.caucho.server.security.XmlAuthenticator">
    <init>
      <user>Harry Potter:quidditch:user</user>
      <password-digest>none</password-digest>
    </init>
  </authenticator>

  <login-config auth-method="basic"/>

  <security-constraint url-pattern="/users-only/*" role-name="user"/>

  ...

</web-app>
```

Using the XmlAuthenticator

In the above example, the <security-constraint> checks for authorization. Only users playing the "user" role can access the /users-only directory.

Another often used authenticator is the JdbcAuthenticator, which uses usernames, passwords, and roles stored in a database.

```
<web-app xmlns="http://caucho.com/ns/resin">

  ...

  <!-- Resin-specific JdbcAuthenticator -->
  <authenticator type='com.caucho.server.security.JdbcAuthenticator'>
    <init>
      <data-source>test</data-source>
      <password-query>
        SELECT password FROM LOGIN WHERE username=?
      </password-query>
      <cookie-auth-query>
        SELECT username FROM LOGIN WHERE cookie=?
      </cookie-auth-query>
      <cookie-auth-update>
        UPDATE LOGIN SET cookie=? WHERE username=?
      </cookie-auth-update>
      <role-query>
        SELECT role FROM LOGIN WHERE username=?
      </role-query>
    </init>
  </authenticator>

  <login-config auth-method='basic'/>

  <security-constraint url-pattern='/users-only/*' role-name='user'/>

  ...

</web-app>
```

login-config

Configures the login class. The web.xml configuration describes the configuration in more detail.

The login can be customized by selecting the `com.caucho.server.security.AbstractLogin` . The `type` attribute will select that class. More sophisticated applications may want to add their own custom AbstractLogin class to replace the predefined values.

Typically a custom login would only be necessary if the application needed a custom way of extracting credentials from the request.

auth-method

Selects the authentication method.

AUTH-METHOD	MEANING
basic	HTTP Basic authentication
digest	HTTP Digest authentication
form	Form-based authentication

auth-method values

form-login-config

Configures authentication for forms. The login form has specific parameters that the servlet engine's login form processing understands. If the login succeeds, the user will see the original page. If it fails, she will see the error page.

form-login-page	The page to be used to prompt the user login	none
form-error-page	The error page for unsuccessful login	none
internal-forward	Use an internal redirect on success or a sendRedirect	false
form-uri-priority	If true, the form's j_uri will override a stored URI	false

The form itself must have the action `j_security_check` . It must also have the parameters `j_username` and `j_password` . Optionally, it can also have `j_uri` and `j_use_cookie_auth` . `j_uri` gives the next page to display when login succeeds. `j_use_cookie_auth` allows Resin to send a persistent cookie to the user to make following login easier.

`j_use_cookie_auth` gives control to the user whether to generate a persistent cookie. It lets you implement the "remember me" button. By default, the authentication only lasts for a single session.

PARAMETER	**MEANING)**
j_username	The user name
j_password	The password
j_uri	Resin extension for the successful display page (Optional).
j_use_cookie_auth	Resin extension to allow cookie login (Optional).

j_security_check Parameters

The following is an example of a servlet-standard login page:

```
<form action='j_security_check' method='POST'>
<table>
<tr><td>User:<td><input name='j_username'>
<tr><td>Password:<td><input name='j_password'>
<tr><td colspan=2>hint: the password is 'quidditch'
<tr><td><input type=submit>
</table>
</form>
```

authenticator

Specifies a class to authenticate users. This Resin-specific option lets you control your authentication. You can either create your own custom authenticator, or use Resin's JdbcAuthenticator.

The authenticator is responsible for taking the username and password and returning a UserPrincipal if the username and password match.

Users wanting to implement an authenticator should look at the JavaDoc for and . To protect your application from API changes, you should extend AbstractAuthenticator rather than implementing Authenticator directly.

XmlAuthenticator

The XmlAuthenticator (com.caucho.serer.security.XmlAuthenticator), stores the authentication in either an xml file or in the configuration itself.

When configuring the XmlAuthenticator in the resin.xml (or web.xml), each `user` adds a new configured user. The value contains the username, password, and the roles the user plays.

```
<authenticator type="com.caucho.server.security.XmlAuthenticator">
  <init>
    <user>Harry Potter:quidditch:user,gryffindor</user>
    <user>Draco Malfoy:pureblood:user,slytherin</user>
    <password-digest>none</password-digest>
  </init>
</authenticator>
```

XmlAuthenticator in resin.xml

Because the plain text passwords in the example above are a serious security issue, most sites will use the password-digest attribute described below to protect the passwords.

ATTRIBUTE	MEANING	DEFAULT
user	specifies an allowed user. May be repeated.	none
password-digest	selects the signature method to protect the password	md5-base64
path	specifies a path to an XML file containing the users and passwords.	none
logout-on-session-timeout	If true, the user will be logged out when the session times out	true

The passwords can be specified in a separate *.xml file. The password file looks like:

```
<authenticator>
  <user name='Harry Potter' password='quidditch' roles='gryffindor'/>
  <user name='Draco Malfoy' password='pureblood' roles='slytherin'/>
</authenticator>
```

password.xml

Sites should use password-digest to protect the passwords.

JdbcAuthenticator

The JdbcAuthenticator () asks a backend database for the password matching the user's name. It uses the DataSource specified by the `pool-name` option, or the JNDI `java:comp/env/jdbc/db-pool` by default. `pool-name` refers to a DataSource configured with database.

The following are the attributes for the JdbcAuthenticator:

ATTRIBUTE	MEANING	DEFAULT
data-source	The database pool. Looks in the application attributes first, then in the global database pools.	none
password-query	A SQL query to get the user's password. The default query is given below.	see below
cookie-auth-query	A SQL query to authenticate the user by a persistent cookie.	none
cookie-auth-update	A SQL update to match a persistent cookie to a user.	none
role-query	A SQL query to determine the user's role. By default, all users are in role "user", but no others.	none
password-digest	Specifies the digest algorithm and format (Resin 2.0.4)	md5-base64
logout-on-session-timeout	If true, the user will be logged out when the session times out (Resin 2.0.6)	true

```
<web-app xmlns="http://caucho.com/ns/resin">

  ...

  <!-- Resin-specific JdbcAuthenticator -->
  <authenticator type='com.caucho.server.security.JdbcAuthenticator'>
    <init>
      <data-source>test</data-source>
      <password-query>
        SELECT password FROM LOGIN WHERE username=?
      </password-query>
      <cookie-auth-query>
        SELECT username FROM LOGIN WHERE cookie=?
      </cookie-auth-query>
      <cookie-auth-update>
        UPDATE LOGIN SET cookie=? WHERE username=?
      </cookie-auth-update>
      <role-query>
        SELECT role FROM LOGIN WHERE username=?
      </role-query>
    </init>
  </authenticator>

  <login-config auth-method='basic'/>

  <security-constraint url-pattern='/users-only/*' role-name='user'/>

  ...

</web-app>
```

LdapAuthenticator

The LdapAuthenticator () uses jndi to contact an LDAP (or Active Directory) server for authentication purposes.

ATTRIBUTE	MEANING	DEFAULT
dn-prefix	string to prepend to query before portion selecting user by name	none
dn-suffix	string to append to query after portion selecting user by name	none
jndi-env	Add a property to the jndi provider used for connecting to the ldap server	see below

logout-on-session-timeout	If true, the user will be logged out when the session times out	true
security-authentication	Sets the Context.SECURITY_AUTHI for the ldap environment	
security-principal	Sets the Context.SECURITY_PRINC for the ldap environment	
security-credentials	Sets the Context.SECURITY_CREDI for the ldap environment	
password-digest	selects the signature method to protect the password	md5-base64
user-attribute	the attribute name to use in the query for matching the user	uid
password-attribute	the attribute name to use for obtaining the password	userPassword
url	the url for the server (since Resin 3.1.1)	ldap://localhost:389

```
<web-app xmlns="http://caucho.com/ns/resin">

  ...

  <authenticator>
    <type>com.caucho.server.security.LdapAuthenticator</type>
    <init>
      <url>ldap://localhost:389</url>
      <dn-suffix>dc=hogwarts,dc=com</dn-suffix>
      <password-digest>none</password-digest>
    </init>
  </authenticator>

  ...

</web-app>
```

jndi-env jndi-env configures properties of the ldap provider implementation. Prior to 3.1.1, the url of the server is specified with `jndi-env` and the `java.naming.provider.url` property.

```
<authenticator>
  <type>com.caucho.server.security.LdapAuthenticator</type>
  <init>
    <jndi-env java.naming.factory.initial="com.sun.jndi.ldap.LdapCtxFactory"/>
    <jndi-env java.naming.provider.url="ldap://localhost:389"/>

    <dn-suffix>dc=hogwarts,dc=com</dn-suffix>
    <password-digest>none</password-digest>
  </init>
</authenticator>
```

LdapAuthenticator jndi-env

JaasAuthenticator

The JaasAuthenticator () uses a JAAS LoginModule for authentication. The JaasAuthenticator is an adapter that provides the ability to use the large number of JAAS LoginModule's included in the JDK for authentication purposes.

ATTRIBUTE	MEANING	DEFAULT
init-param	Add a property to the LoginModule	none
login-module	The fully qualified class name of the LoginModule implementation	required
logout-on-session-timeout	If true, the user will be logged out when the session times out	true
password-digest	selects the signature method to protect the password	md5-base64

```
<web-app xmlns="http://caucho.com/ns/resin">

  <authenticator type="com.caucho.server.security.JaasAuthenticator">
    <init>
      <login-module>com.sun.security.auth.module.Krb5LoginModule</login-module>
      <init-param>
        <debug>true</debug>
      </init-param>
    </init>
  </authenticator>

</web-app>
```

JaasAuthenticator configuration

isUserInRole The isUserInRole method is supported if the LoginModule provides either an isUserInRole method in the Principal returned by the LoginModule, or a getRoles() method returning a java.util.Set. (Since 3.0.19).

init-param <init-param> directives are used to configure the properties of the LoginModule. Existing LoginModules provide documentation of the init-param that are accepted. Custom LoginModule implementations retrieve the init-param values in the initialize method.

Custom LoginModule

```
import java.util.*;

import javax.security.auth.*;
import javax.security.auth.spi.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;

public class TestLoginModule implements javax.security.auth.spi.LoginModule {
  private Subject _subject;
  private CallbackHandler _handler;
  private Map _state;

  private String _userName;
  private String _password;

  public void initialize(Subject subject,
                         CallbackHandler handler,
                         Map sharedState,
                         Map options)
   {
     _subject = subject;
     _handler = handler;
     _state = sharedState;

     _userName = (String) _options.get("user");
     _password = (String) _options.get("password");
   }

   public boolean login()
     throws LoginException
   {
     NameCallback name = new NameCallback("");
     PasswordCallback password = new PasswordCallback("", false);

     _handler.handle(new Callback[] { name, password });

     if (_userName.equals(name.getName()) &&
         _password.equals(password.getPassword()) {
         _subject.getPrincipals().add(new TestPrincipal(_userName));
       return true;
     }
     else
       return false;
   }

   public boolean abort()
   {
     return true;
   }

   public boolean commit()
   {
     return _subject.getPrincipals().size() > 0;
   }

   public boolean logout()
   {
      return true;
   }
}
```

Custom LoginModule - java code

```
<web-app xmlns="http://caucho.com/ns/resin">

  <authenticator type="com.caucho.server.security.JaasAuthenticator">
    <init>
      <login-module>example.TestModule</login-module>
      <init-param>
        <user>Harry</user>
        <password>quidditch</password>
      </init-param>
    </init>
  </authenticator>
</web-app>
```

Custom LoginModule - resin-web.xml configuration

AuthenticationList

AuthenticatorList () is used to configure more than one authenticator in a list, each authenticator is tried in turn and if the authentication fails the next authenticator in the list is attempted.

```
  <authenticator type="com.caucho.server.security.AuthenticatorList">
    <init>
      <authenticator resin:type="com.caucho.server.security.XmlAuthenticator">
        <user>admin:NIHlOSafJN2H7emQCkOQ2w==:user,admin</user>
      </authenticator>

      <authenticator resin:type='com.caucho.server.security.JdbcAuthenticator'>
        <data-source>jdbc/users</data-source>
        <password-query>
          SELECT password FROM LOGIN WHERE username=?
        </password-query>
        <cookie-auth-query>
          SELECT username FROM LOGIN WHERE cookie=?
        </cookie-auth-query>
        <cookie-auth-update>
          UPDATE LOGIN SET cookie=? WHERE username=?
        </cookie-auth-update>
        <role-query>
          SELECT role FROM LOGIN WHERE username=?
        </role-query>
      </authenticator>
    </init>
  </authenticator>

  <login-config auth-method='basic'/>

  <security-constraint url-pattern='/users/*' role-name='user'/>
  <security-constraint url-pattern='/admin/*' role-name='admin'/>
```

Digest passwords

Digest protects passwords

Digest passwords enable an application to avoid storing and even transmitting the password in a form that someone can read.

A digest of a cleartext password is calculated when it is passed through a one-way function that consistently produces another series of characters, `digestPassword = digester(username + ":" + realm + ":" cleartextPassword)` . The function is "one-way" because the digestPassword cannot be used to reverse-engineer the original password.

Digest passwords can be used in two places: storage and transmission. Digest passwords in storage means that the password is stored in a digested form, for example in a database or in a file. Digest passwords in transmission means that the client (usually a web browser) creates the digest and submits the digest password to the web server.

Storing digest passwords is so important for security purposes that the Resin authenticators default to assuming that the passwords are stored in digest form.

The important advantage is that a user's cleartext password is not as easily compromised. Since the password they use (the "cleartext" password) is not stored a malicious user cannot determine the password by gaining access to the database or other backend storage for the passwords.

MD5 digest

Resin's authenticators use "MD5-base64" and a realm "resin" to digest passwords by default. `MD5` indicates that the MD5 algorithm is used. `base64` is an encoding format to apply to the binary result of MD5.

Some examples are:

USERNAME	REALM	PASSWORD	DIGEST
root	resin	changeme	j/qGVP4C0T7UixSpKJpTdw==
harry	resin	quidditch	uTOZTGaB6pooMDvqvl2Lbg==
hpotter	resin	quidditch	x8i6aM+zOwDqqKPRO/vkxg==
filch	resin	mrsnorris	KmZIq2RKXAHV4BaoNHfupQ==
pince	resin	quietplease	Txpd1jQc/xwhISIqodEjfw==
snape	resin	potion	I7HdZr7CTM6hZLlSd2o+CA==
mcgonagall	resin	quidditch	4slsTREVeTo0sv5hGkZWag==
dmalfoy	resin	pureblood	yI2uN1l97Rv5E6mdRnDFwQ==
lmalfoy	resin	myself	sj/yhtU1h4LZPw7/Uy9IVA==

In the above example the digest of "harry/quidditch" is different than the digest of "hpotter/quidditch" because even though the password is the same, the username has changed. The digest is calculated with `digest(username +`

`":" + realm + ":" + password)` , so if the username changes the resulting digest is different.

Calculating a digest

Of course, storing the digest password is a bit more work. When the user registers, the application needs to compute the digest to store it.

Unix users can quickly calculate a digest:

```
echo -n "user:resin:password" | openssl dgst -md5 -binary | uuencode -m -
```

The class can be used to calculate a digest.

```
import com.caucho.server.security.PasswordDigest;

...

String username = ...;
String password = ...;
String realm = "resin";

PasswordDigest passwordDigest = PasswordDigest();

String digest = passwordDigest.getPasswordDigest(username, password, realm);
```

Calculating a digest - Java example

```
$username = ...;
$password = ...;
$realm = "resin";

$passwordDigest = new Java("com.caucho.server.security.PasswordDigest");

$digest = $passwordDigest->getPasswordDigest($username, $password, $realm);
```

Calculating a digest - PHP example

The realm for JdbcAuthenticator and XmlAuthenticator defaults to "resin"; the realm can be specified during configuration:

```
<authenticator type='com.caucho.server.security.JdbcAuthenticator'>
  <init>
    <password-digest-realm>hogwarts</password-digest-realm>

    ...
```

Specifying a realm

Using Digest with basic authentication or a form login

When using the form login method or the HTTP basic authentication login method, the password submitted is in cleartext. The Resin authenticator will digest the password before comparing it to the value retrieved from storage. The message is transmitted in cleartext but is stored as a digest. This method provides only half of the protection - the password is not protected in transmission (although if the form submit is being done over an SSL connection it will be secure).

Using HTTP digest authentication

The HTTP protocol includes a method to indicate to the client that it should make a digest using the password. The client submits a digest to Resin instead of submitting a cleartext password. HTTP digest authentication protects the password in transmission.

When using HTTP digest, Resin will respond to the browser and ask it to calculcate a digest. The steps involved are:

- Resin provides the client a realm and some other information
- The client obtains a username and password (usually a dialog box with a web browser)
- The client calculates a digest using the username, realm, pasword, and other information supplied by Resin
- The client submits the digest to Resin
- Resin does the same digest calculation as the client did
- Resin compares the submitted digest and the digest it calculated. If they match, the user has been authenticated

The advantage of this method is that the cleartext password is protected in transmission, it cannot be determined from the digest that is submitted by the client to the server.

HTTP digest authentication is enabled with the child of the configuration tag.

```
<login-config>
  <auth-method>DIGEST</auth-method>
</login-config>
```

Using HTTP digest authentication

Disabling the use of password-digest

Although it is not advised, Resin's authenticators can be configured to use passwords that are not in digest form.

```
<authenticator>
  <type>com.caucho.server.security.XmlAuthenticator</type>
  <init>
     <password-digest>none</password-digest>
     <user>harry:quidditch:user</user>
  </init>
</authenticator>
```

Disabling the use of password-digest

Compatibility

Authenticators are not defined by the Servlet Specification, so the ability to use passwords stored as a digest depends upon the implementation of the Authenticator that the application server provides. MD5-base64 is the most common form of digest, because it is the default in HTTP digest authentication.

The use of <auth-method>DIGEST<auth-method> is defined in the Servlet Specification and implemented in most application servers.

Single Signon

"Single signon" refers to allowing for a single login for more than one context, for example, logging in to all web-apps in a server at once. You can implement single signon by configuring the authenticator in the proper environment: web-app, host, or server. The login will last for all the web-apps in that environment.

The authenticator is a resource which is shared across its environment. For example, to configure the XML authenticator for all web-apps in foo.com, you might configure as follows:

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">
    <http port="8080"/>

    <host id="foo.com">
      <root-directory>/opt/foo.com</root-directory>

      <authenticator type="com.caucho.server.security.XmlAuthenticator">
        <init>
          <!-- password: quidditch -->
          <user>harry:uTOZTGaB6pooMDvqvl2LBu:user,gryffindor</user>
          <!-- password: pureblood -->
          <user>dmalfoy:yI2uN1l97Rv5E6mdRnDFDB:user,slytherin</user>
        </init>
      </authenticator>

      <web-app-deploy path="webapps"/>
    </host>
  </cluster>
</resin>
```

Single Signon for foo.com

Any .war in the webapps directory will share the same signon for the host. You will still need to implement a login-config for each web-app.

The value of reuse-session-id must be `true` for single signon.

Single signon for virtual hosts

The basis for establishing client identity is the JSESSIONID cookie. If single signon is desired for virtual hosts, Resin must be configured to notify the browser of the proper domain name for the cookie so that the same JSESSIONID cookie is submitted by the browser to each virtual host.

The authenticator is placed at the cluster level so that it is common to all virtual hosts. The cookie-domain is placed in a web-app-default at the cluster level so that it is applied as the default for all webapps in all virtual hosts.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">
    <http port="8080"/>

    <authenticator type="com.caucho.server.security.XmlAuthenticator">
      ...
    </authenticator>

    <web-app-default>
      <session-config>
        <enable-url-rewriting>false</enable-url-rewriting>
        <cookie-domain>.hogwarts.com</cookie-domain>
      </session-config>
    </web-app-default>


    <host id="gryffindor.hogwarts.com">
      ...
    </host>

    <host id="slytherin.hogwarts.com">
      ...
    </host>
  </server>
</resin>
```

Single Signon for gryffindor.hogwarts.com and slytherin.hogwarts.com

Because of the way that browsers are restricted by the HTTP specification from submitting cookies to servers, it is not possible to have a single signon for virtual hosts that do not share some portion of their domain name. For example, "gryffindor.com" and "slytherin.com" cannot share a common authentication.

Custom Login

The Login is primarily responsible for extracting the credentials from the request (typically username and password) and passing those to the ServletAuthenticator.

The Servlet API calls the Login in two contexts: directly from `ServletRequest.getUserPrincipal()` , and during security checking. When called from the Servlet API, the login class can't change the response. In other words, if an application calls getUserPrincipal(), the Login class can't return a forbidden error page. When the servlet engine calls authenticate(), the login class can return an error page (or forward internally.)

Normally, Login implementations will defer the actual authentication to a ServletAuthenticator class. That way, both "basic" and "form" login can use the same JdbcAuthenticator. Some applications, like SSL client certificate login, may want to combine the Login and authentication into one class.

Login instances are configured through bean introspection. Adding a public `setFoo(String foo)` method will be configured with the following login-config:

```
<login-config type="test.CustomLogin">
  <init>
    <foo>bar</bar>
  </init>
</login-config>
```

Security Constraints

security-constraint

child of: web-app

Selects protected areas of the web site. Sites using authentication as an optional personalization feature will typically not use any security constraints. Sites using authentication to limit access to certain sections of the website to certain users will use security constraints.

Security constraints can also be custom classes.

```
<web-app>
  ...
<security-constraint>
  <web-resource-collection>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint role-name='user'/>
</security-constraint>
  ...
</web-app>
```

Protecting all pages for logged-in users

web-resource-collection

child of: security-constraint

Specifies a collection of areas of the web site.

url-pattern	url patterns describing the resource
http-method	HTTP methods to be restricted.

auth-constraint

child of: security-constraint

Requires that authenticated users fill the specified role. In Resin's JdbcAuthenticator, normal users are in the "user" role. Think of a role as a group of users.

role-name	Roles which are allowed to access the resource.

```
<security-constraint>
  <auth-constraint role-name='webdav'/>

  <web-resource-collection>
    <url-pattern>/webdav/*</url-pattern>
  </web-resource-collection>
</security-constraint>
```

Protecting webdav for webdav users

ip-constraint

child of: security-constraint

Allow or deny requests based on the ip address of the client. ip-constraint is very useful for protecting administration resources to an internal network. It can also be useful for denying service to known problem ip's.

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>

  <ip-constraint>
    <allow>192.168.17.0/24</allow>
  </ip-constraint>
</security-constraint>
```

Admin pages allowed from 192.168.17.0/24

The `/24` in the ip `192.168.17.0/24` means that the first 24 bits of the ip are matched - any ip address that begins with `192.168.17.` will match. The usage of `/bits` is optional.

```
<security-constraint>
  <ip-constraint>
    <deny>205.11.12.3</deny>
    <deny>213.43.62.45</deny>
    <deny>123.4.45.6</deny>
    <deny>233.15.25.35</deny>
    <deny>233.14.87.12</deny>
  </ip-constraint>

  <web-resource-collection>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
</security-constraint>
```

Block out known trouble makers

Be careful with deny - some ISP's (like AOL) use proxies and the ip of many different users may appear to be the same ip to your server.

allow	add an ip address to allow	default is to allow all ip addresses
deny	add an ip address to deny	default is to deny no ip addresses
error-code	error code to send if request is denied	403
error-message	error message to send if request is denied	Forbidden IP Address

cache-size	cache size, the result of applying rules for an ip is cached for subsequent requests	256

If only `deny` is used, then all ip's are allowed if they do not match a `deny` . If only `allow` is used, then an ip is denied unless it matches an `allow` . If both are used, then the ip must match both an `allow` and a `deny`

user-data-constraint

child of: security-constraint

Restricts access to secure transports, i.e. SSL.

transport-guarantee	Required transport properties. NONE, INTEGRAL, and CONFIDENTIAL are allowed values.

```
<security-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>

  <web-resource-collection>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
</security-constraint>
```

The default behaviour is for Resin to rewrite any url that starts with "http:" by replacing the "http:" part with "https:", and then send redirect to the browser.

If the default rewriting of the host is not appropriate, you can set the for the host:

```
  <host id='...'>
    <secure-host-name>https://hogwarts.com</secure-host-name>

 ...
```

```
  <host id='...'>
    <secure-host-name>https://hogwarts.com:8443</secure-host-name>

  ...
```

transport-guarantee

Restricts access to secure transports, i.e. SSL.

constraint

child of: security-constraint

Defines a custom constraint. The custom constraint specifies a `resin:type` which extends . Bean-style initialization is used to initialize the constraint.

```
...
<security-constraint>
  <constraint resin:type="example.CustomConstraint>
    <init>
      <policy>strict</policy>
    </init>
  </constraint>
  <web-resource-collection url-pattern='/*'/>
</security-constraint>
...
```

Custom Security Constraints

Any custom security constraint is checked after any authentication (login) but before any filters or servlets are applied. The security constraint will return true if the request is allowed and false if it's forbidden. If the request is forbidden, it's the constraint's responsibility to use response.sendError() or to return an error page.

```
package example;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.caucho.server.security.*;

public class CustomSecurity extends AbstractConstraint {
  private String foo = "false";

  public void setFoo(String foo)
  {
    this.foo = foo;
  }

  public boolean needsAuthentication()
    return false;
  }

  public boolean isAuthorized(HttpServletRequest request,
                              HttpServletResponse response,
                              ServletContext application)
    throws ServletException, IOException
  {
    if (foo.equals(request.getParameter("test")))
      return true;

    response.sendError(response.SC_FORBIDDEN);

    return false;
  }
}
```

The `needsAuthentication` method tells Resin that it needs to log in the user before checking the authorization. This would allow the custom authorizer to check user roles or the user principle for the proper permissions.

```
<constraint resin:type="example.CustomSecurity">
  <foo>test-value</foo>
</constraint>
```

Protecting static files from viewing by anyone

Sometimes it is necessary to protect files from being viewed by anyone, such as configuration files used in your code but not meant to be served to a browser.

Place files in WEB-INF

Place files in `WEB-INF` or a subdirectory of `WEB-INF` . Any files in `WEB-INF` or it's subdirectories will automatically be protected from viewing.

Security constraint requiring role nobody

Use a security constraint that requires a `role` that nobody will ever have.

```
<web-app>
  ...
  <!-- protect all .properties files -->
  <security-constraint>
    <web-resource-collection>
      <url-pattern>*.properties</url-pattern>
    </web-resource-collection>
    <auth-constraint role-name='nobody'/>
  </security-constraint>

  <!-- protect the config/ subdirectory -->
  <security-constraint>
    <web-resource-collection>
      <url-pattern>/config/*</url-pattern>
    </web-resource-collection>
    <auth-constraint role-name='nobody'/>
  </security-constraint>
  ...
</web-app>
```

security-constraint to protect static files

A servlet that returns a 403 error

Use a simple servlet that returns a 403 error, which means "Forbidden". Resin provides the servlet which is useful for this:

```
<web-app>
  ...
  <servlet>
    <servlet-name>forbidden</servlet-name>
    <servlet-class>com.caucho.servlets.ErrorStatusServlet</servlet-class>
    <init>
      <status-code>403</status-code>
    </init>
  </servlet>

  <servlet-mapping url-pattern="*.properties" servlet-name="forbidden"/>
  <servlet-mapping url-pattern="/config/*" servlet-name="forbidden"/>
  ...
</web-app>
```

Using ErrorStatusServlet to protect static files

Or you could implement your own servlet:

```
package example.servlets;

import javax.servlet.*;
import javax.servlet.http.*;

import java.io.IOException;

/**
 * Respond with a 403 error
 */
public class Forbidden extends GenericServlet {
  public void service(ServletRequest request, ServletResponse response)
    throws ServletException, IOException
  {
    HttpServletResponse res = (HttpServletResponse) response;
    res.sendError(403);
  }
}
```

servlet to protect static files - WEB-INF/classes/example/servlets/Forbidden.java

What SSL provides

SSL provides two kinds of protection, `encryption` and `server authentication` .

Encryption

public key
A set of bytes used to `encrypt` data and `verify signatures` . The key is public because it can be made available without a loss of security. The public key can only be used for encryption; it cannot decrypt anything. A public key always has a corresponding `private key` .

SSL provides encryption of the data traffic betweeen a client and a server. When the traffic is encrypted, an interception of that traffic will not reveal the contents because they have been encrypted - it will be unusable nonsense.

private key
A set of bytes used to `decrypt` data and `generate signatures` . The key is private because it must be kept secret or there will be a loss of security. The private key is used for decryption of data that has been encrypted with the corresponding `public key` .

SSL uses public key cryptography. Public key cryptography is based upon a pair of keys, the public key and the private key. The public key is used to encrypt the data. Only the corresponding private key can successfully decrypt the data.

For example, when a browser connects to Resin, Resin provides the browser a public key. The browser uses the public key to encrypt the data, and Resin uses the private key to decrypt the data. For this reason, it is important that you never allow anyone access to the private key, if the private key is obtained by someone then they can use it to decrypt the data traffic.

Encryption is arguably the more important of the security meausures that SSL provides.

Server Authentication

certificate
A combination of a `private key` , identity information (such as company name), and a `signature` generated by a `signing authority` . `private key` .

SSL also provides the ability for a client to verify the identity of a server. This is used to protect against identity theft, where for example a malicious person imitates your server or redirects client traffic to a different server while pretending to be you.

signing authority
A company that is trusted to sign certificates. Browsers include certificates of signing authorities that they trust.

Server authentication uses the signature aspect of public key cryptography. The private key is used to sign messages, and the public key is used to verify the

signature. With SSL, the validity of signatures depends upon signing authorities. Signing authorites (also called certificate authorities) are companies who have generated public keys that are included with browser software. The browser knows it can trust the signing authority, and the signing authority signs your SSL certificate, putting its stamp of approval on the information in your certificate.

`certificate authority`
Another name for `signing authority` . A company that is trusted to sign certificates. Browsers include certificates of signing authorities that they trust.

For example, after you generate your public and private key, you then generate a signing request and send it to a signing authority. This signing request contains information about your identity, this identity information is confirmed by the signing authority and ultimately displayed to the user of the browser. The signing authority validates the identity information you have provided and uses their private key to sign, and then returns a `certificate` to you. This certificate contains the identity information and your public key, verified by the signing authority, and is provided to the browser. Since the browser has the public key of the signing authority, it can recognize the signature and know that the identity information has been provided by someone that can be trusted.

OpenSSL

OpenSSL is the same SSL implementation that Apache's mod_ssl uses. Since OpenSSL uses the same certificate as Apache, you can get signed certificates using the same method as for Apache's mod_ssl or following the OpenSSL instructions.

Linking to the OpenSSL Libraries on Unix

On Unix systems, Resin's libexec/libresinssl.so JNI library supports SSL using the OpenSSL libraries. Although the ./configure script will detect many configurations, you can specify the openssl location directly:

```
resin> ./configure --with-openssl=/usr/local/ssl
```

Obtaining the OpenSSL Libraries on Windows

On Windows systems, the resinssl.dll includes JNI code to use OpenSSL libraries (it was in resin.dll in versions before 3.0). All you need to do is to obtain an OpenSSL binary distribution and install it.

Resin on Windows is compiled against the GnuWin32 binary, you can obtain an installation package here.

Once you have run the installation package, you can copy the necessary dll libraries into `$RESIN_HOME` :

```
C:\> cd %RESIN_HOME%
C:\resin-3.0> copy "C:\Program Files\GnuWin32\bin\libssl32.dll" .\libssl32.dll
C:\resin-3.0> copy "C:\Program Files\GnuWin32\bin\libeay32.dll" .\libeay32.dll
```

Copying the Windows SSL libraries into $RESIN_HOME

Preparing to use OpenSSL for making keys

You can make a `keys/` subdirectory of $RESIN_HOME to do your work from and as a place to store your generated keys.

```
unix> cd $RESIN_HOME
unix> mkdir keys
unix> cd keys

win> cd %RESIN_HOME%
win> mkdir keys
win> cd keys
```

$RESIN_HOME/keys

Using OpenSSL requires a configuration file. Unix users might find the default configuration file in `/usr/ssl/openssl.cnf` or `/usr/share/ssl/openssl.cnf` . Windows users may not have received one with their package.

Either way, it can be valuable to make your own `openssl.cnf` that is used just for generating the keys to use with Resin. You can use the following as a template for a file `$RESIN_HOME/keys/openssl.cnf` . You may want to fill in the `_default` values so you don't have to type them in every time.

```
[ req ]
 default_bits            = 1024
 distinguished_name      = req_distinguished_name

[ req_distinguished_name ]
 C                       = 2 letter Country Code, for example US
 C_default               =
 ST                      = State or Province
 ST_default              =
 L                       = City
 L_default               =
 O                       = Organization Name
 O_default               =
 OU                      = Organizational Unit Name, for example 'Marketing'
 OU_default              =
 CN                      = your domain name, for example www.hogwarts.com
 CN_default              =
 emailAddress            = an email address
 emailAddress_default    =
```

$RESIN_HOME/keys/openssl.cnf

Creating a private key

Create a private key for the server. You will be asked for a password - don't forget it! You will need this password anytime you want to do anything with this private key. But don't pick something you need to keep secret, you will need to put this password in the Resin configuration file.

```
unix> openssl genrsa -des3 -out gryffindor.key 1024
win>  "C:\Program Files\GnuWin32\bin\openssl.exe" \
         genrsa -des3 -out gryffindor.key 1024
```

creating the private key gryffindor.key

Creating a certificate

OpenSSL works by having a signed public key that corresponds to your private key. This signed public key is called a `certificate` . A certificate is what is sent to the browser.

You can create a self-signed certificate, or get a certificate that is signed by a certificate signer (CA).

Creating a self-signed certificate You can create a certificate that is self-signed, which is good for testing or for saving you money. Since it is self-signed, browsers will not recognize the signature and will pop up a warning to browser users. Other than this warning, self-signed certificates work well. The browser

cannot confirm that the server is who it says it is, but the data between the browser and the client is still encrypted.

```
unix> openssl req -config ./openssl.cnf -new -key gryffindor.key \
        -x509 -out gryffindor.crt
win> "C:\Program Files\GnuWin32\bin\openssl.exe" req -config ./openssl.cnf \
         -new -key gryffindor.key -x509 -out gryffindor.crt
```

creating a self-signed certificate gryffindor.crt

You will be asked to provide some information about the identity of your server, such as the name of your Organization etc. Common Name (CN) is your domain name, like: "www.gryffindor.com".

Creating a certificate request To get a certificate that is signed by a CA, first you generate a `certificate signing request` (CSR).

```
unix> openssl req -new -config ./openssl.cnf -key gryffindor.key \
      -out gryffindor.csr
win> "C:\Program Files\GnuWin32\bin\openssl.exe" req -new \
      -config ./openssl.cnf  -key gryffindor.key -out gryffindor.csr
```

creating a certificate request gryffindor.csr

You will be asked to provide some information about the identity of your server, such as the name of your Organization etc. Common Name (CN) is your domain name, like: "www.gryffindor.com".

Send the CSR to a certificate signer (CA). You'll use the CA's instructions for Apache because the certificates are identical. Some commercial signers include:

- Verisign
- Thawte Consulting

You'll receive a *gryffindor.crt* file.

Most browsers are configured to recognize the signature of signing authorities. Since they recognize the signature, they will not pop up a warning message the way they will with self-signed certificates. The browser can confirm that the server is who it says it is, and the data between the browser and the client is encrypted.

resin.xml - Configuring Resin to use your private key and certificate

The OpenSSL configuration has two tags and . These correspond exactly to mod_ssl's SSLCertificateFile and SSLCertificateKeyFile. So you can use the same certificates (and documentation) from mod_ssl for Resin.

The full set of parameters is in the port configuration.

```
...
<http port="443">
  <openssl>
    <certificate-file>keys/gryffindor.crt</certificate-file>
    <certificate-key-file>keys/gryffindor.key</certificate-key-file>
    <password>my-password</password>
  </openssl>
</http>
```

Testing

Testing with the browser A quick test is the following JSP.

```
Secure? <%= request.isSecure() %>
```

Using openssl to test the server The openssl tool can be used as a client, showing some interesting information about the conversation between the client and the server:

```
unix$ openssl s_client -connect www.some.host:443 -prexit
```

Certificate Chains

A `certificate chain` is used when the signing authority is not an authority trusted by the browser. In this case, the signing authority uses a certificate which is in turn signed by a trusted authority, giving a chain of `[your certificate] <--- signed by ---- [untrusted signer] <---- signed by ---- [trusted signer]` .

The Resin config parameter is used to specify a certificate chain. It is used to reference a file that is a concatenation of:

1. your certificate file
2. the intermediate (untrusted) certificate
3. the root (trusted) certificate.

The certificates must be in that order, and must be in PEM format.

Example certificate chain for Instant SSL Comodo (http://instantssl.com) is a signing authority that is untrusted by most browsers. Comodo has their certificate signed by GTECyberTrust.

Comodo gives you three certificates:

1. `your_domain.crt` (signed by Comodo)
2. `ComodoSecurityServicesCA.crt` (signed by GTE CyberTrust)
3. `GTECyberTrustRoot.crt` (universally known root)

In addition to this, you have your key, `your_domain.key` . The contents of the file referred to by is a concatenation of the three certificates, in the correct order.

```
$ cat your_domain.crt ComodoSecurityServicesCA.crt GTECyberTrustRoot.crt > chain.txt
```

Creating a certificate chain file

```
<http port="443">
  <openssl>
    <certificate-key-file>keys/your_domain.key</certificate-key-file>
    <certificate-file>keys/your_domain.crt</certificate-file>
    <certificate-chain-file>keys/chain.txt</certificate-chain-file>
    <password>test123</password>
  </openssl>
</http>
```

resin.xml using a certificate chain file

JSSE

We recommend avoiding JSSE if possible. It is slower than using Resin's OpenSSL support and does not appear to be as stable as Apache or IIS (or Netscape/Zeus) for SSL support. In addition, JSSE is far more complicated to configure. While we've never received any problems with Resin using OpenSSL, or SSL from Apache or IIS, JSSE issues are fairly frequent.

Install JSSE from Sun

This section gives a quick guide to installing a test SSL configuration using Sun's JSSE. It avoids as many complications as possible and uses Sun's keytool to create a server certificate.

Resin's SSL support is provided by Sun's JSSE. Because of export restrictions, patents, etc, you'll need to download the JSSE distribution from Sun or get a commercial JSSE implementation.

More complete JSSE installation instructions for JSSE are at http://java.sun.com/products/jsse/install.html.

1. First download Sun's JSSE.

2. Uncompress and extract the downloaded file.

3. Install the JSSE jar files: jsse.jar, jnet.jar, and jcert.jar. You can either put them into the CLASSPATH or you can put them into $JAVA_HOME/jre/lib/ext. Since you will use "keytool" with the new jars, you need to make them visible to keytool. Just adding them to resin/lib is not enough.

4. Register the JSSE provider (com.sun.net.ssl.internal.ssl.Provider). Modify $JAVA_HOME/jre/lib/security/java.security so it contains something like:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.net.ssl.internal.ssl.Provider
```

   Adding the JSSE provider allows "keytool" to create a key using the RSA algorithm.

Create a test server certificate

The server certificate is the core of SSL. It will identify your server and contain the secret key to make encryption work.

- Sun's keytool
- A self-signed certificate using open_ssl
- A test certificate from Thawte
- A production certificate from one of the certificate authorities (Verisign, Thawte, etc)

In this case, we're using Sun's `keytool` to generate the server certificate. Here's how:

```
resin1.2.b2>  \texttt{mkdir keys}
resin1.2.b2>  \texttt{keytool -genkey -keyalg RSA -keystore keys/server.keystore}
Enter keystore password:   \texttt{changeit}
What is your first and last name?
  [Unknown]:   \texttt{www.caucho.com}
What is the name of your organizational unit?
  [Unknown]:   \texttt{Resin Engineering}
What is the name of your organization?
  [Unknown]:   \texttt{Caucho Technology, Inc.}
What is the name of your City or Locality?
  [Unknown]:   \texttt{San Francisco}
What is the name of your State or Province?
  [Unknown]:   \texttt{California}
What is the two-letter country code for this unit?
  [Unknown]:   \texttt{US}
Is <CN=www.caucho.com, OU=Resin Engineering,
  O="Caucho Technology, Inc.", L=San Francisco, ST=California, C=US> correct?
  [no]:   \texttt{yes}

Enter key password for <mykey>
        (RETURN if same as keystore password):   \texttt{changeit}
```

Currently, the key password and the keystore password must be the same.

resin.xml

The Resin SSL configuration extends the http configuration with a few new elements.

```
<resin xmlns="http://caucho.com/ns/resin">
  <server>

    <http port="8443">
     <jsse-ssl>
       <key-store-type>jks</key-store-type>
       <key-store-file>keys/server.keystore</key-store-file>
       <password>changeit</password>
     </jsse-ssl>
    </http>

    ...

  </http-server>
</caucho.com>
```

Testing JSSE

With the above configuration, you can test SSL with https://localhost:8443. A quick test is the following JSP.

```
Secure? <%= request.isSecure() %>
```

User Experiences with JSSE

How do I configure Resin with SSL using JSSE? Nicholas Lehuen writes:

Here is a short step-by-step installation guide for SSL on Resin.
The purpose : to install SSL support on Resin**Requirements**

- The latest Resin 1.2 snapshot (I used the 08/04/2000 snapshot with success) http://www.caucho.com/download/index.xtp
- JSSE 1.0.1 http://java.sun.com/products/jsse/
- Optional : a certificate authority (CA) such as Verisign, Thawte, or your own. Thawte is providing a free test certificate authority service which enables you to check the certification process before buying your own certificate. Of course, you can also skip the CA by providing self-signed public key certificate. This will be explained later.

  https://www.thawte.com/cgi/server/test.exe

JSSE setup

1. Follow the installation instructions http://java.sun.com/products/jsse/install.html
2. Even if Resin has its own provider registration system (we'll see it on next step), I suggest that you statically register the SunJSSE by editing the <java-home>/lib/security/java.security as explained in the installation guide. This will ease the use of keytool.

Keystore initialization

1. Create a directory named 'keys' somewhere in your Resin installation. I suggest you place it in the Resin home directory.
2. Copy the file <java-home>/lib/security/cacerts into the 'keys' directory
3. Rename the cacerts file as you want. I'll suppose you name it 'private.keystore'.

Keystore protection

Your private.keystore file is for the moment a copy of the cacerts keystore, which contains the CA public key certificates (very important for client HTTPS connections). We will insert your own private key in this file, thus it'll have to be password-protected, so that anyone stealing it will have difficulties in forging certificates on your behalf.

1. Go into the 'keys' directory
2. type the following command : keytool -storepasswd -storepass changeit -new YourPasswordHere \ -keystore private.keystore

(the default password for the cacerts keystore is 'changeit')**Private key generation**

We'll now generate your key pair, which is composed of a private (the one which MUST remain secret !) and a public key. The point here is to use the RSA key pair generator, and NOT the default one, which is DSA. This is were the JSSE security provider is used.

type the following command :

```
M:\keys>keytool -genkey -keyalg RSA -alias myserverkeypair \
                -storepass YourPasswordHere -keystore private.keystore
What is your first and last name?
  [Unknown]:  www.myserver.com
What is the name of your organizational unit?
  [Unknown]:  Foo Dept
What is the name of your organization?
  [Unknown]:  Bar
What is the name of your City or Locality?
  [Unknown]:  Paris
What is the name of your State or Province?
  [Unknown]:  France
What is the two-letter country code for this unit?
  [Unknown]:  FR
Is <CN=www.myserver.com, OU=Foo Dept, O=Bar, L=Paris,
                ST=France, C=FR> correct?
  [no]:  yes

Enter key password for <myserverkeypair>
        (RETURN if same as keystore password):
```

You MUST mention your HTTP server name as the CN of the certificate (thus the reply to 'first and last name'). Browsers would emit warnings to your users if you didn't. Any other informations are at your choice, however the process of key pair generation and attributes definitions is very strict for "real-life" cryptography, i.e. Verisign will double-check your identity, address and so on.

Another important point : DON'T AFFECT A PASSWORD to your key pair. It must remain the same as the keystore, at least until Resin provides a means of configuring the key pair password.**Public Key Certificate (optional)**

Request a public key certificate and insert the public key certificate into your keystore.

For users to trust your server, you'll have to have your public key certificate (PKC) signed by a Certificate Authority (CA) (Verisign, Thawte, Certplus...). This is done by sending a certificate signature request (CSR) to the CA, coping

with all the legal stuff and getting a signed PKC in return. This step is mandatory for production server, unless you have some means to convince your users that your PKC is valid without a CA signature, which is possible in intranet environment for example. However, for testing purpose, you can start by using your self-signed PKC without any CA signature. An intermediary solution is to use a test CA so that you can check that your CSR is correctly emitted, that the Certificate Chain is correctly checked, and so on. Thawte provides a test CA at the address mentioned above.

1. Generate a CSR by typing the following command :

```
M:\keys>keytool -certreq -alias myserverkeypair -storepass YourPasswordHere \
                -keystore private.keystore
-----BEGIN NEW CERTIFICATE REQUEST-----
MIIBqjCCARMCAQAwajELMAkGA1UEBhMCRlIxDzANBgNVBAgTBkZyYW5jZTEOMAwGA1UEBxMFUGFy
... cut ...
KDYZTklbg1NOiXTdXIhPHb3+YOgZ+HoeDTxOx/rRhA==
-----END NEW CERTIFICATE REQUEST-----
```

2. Copy/Paste the CSR into the text box at the following address. Leave all options with their default value. https://www.thawte.com/cgi/server/test.exe

3. You'll get a certificate looking like :

```
-----BEGIN CERTIFICATE-----
MIICjzCCAfigAwIBAgIDBp8SMA0GCSqGSIb3DQEBBAUAMIGHMQswCQYDVQQGEwJa
... cut ...
/93Q58iI4fgQ/kc+l8ogpVwh/IJwlUjmszd19Jf+pxyySMM=
-----END CERTIFICATE-----
```

4. Copy/Paste this certificate into a file named 'myserver.cer' . If you have Microsoft Internet Explorer 5.0 (maybe 4.0) installed, you can open this .cer file and see the certificate as your user will when they ask the security properties of pages served securely by your server. A warning should be emitted, stating that you can't trust the certificate as it does not point to a trusted root CA. You can keep going with this warning or download and trust the test root CA (available on https://www.thawte.com/servertest.crt). Be ware though that the final user should not and surely won't accept to trust this test root CA.

5. Anyway, to be able to import your signed certificate, you'll have to import the test root CA certificate. Download it and import it using the following command :

```
M:\keys>keytool -import -alias servertest -storepass YourPasswordHere \
                -keystore private.keystore -file servertest.crt
Owner: CN=Thawte Test CA Root, OU=TEST, O=Thawte, ST=FOR TESTING, C=ZA
Issuer: CN=Thawte Test CA Root, OU=TEST, O=Thawte, ST=FOR TESTING, C=ZA
Serial number: 0
Valid from: Thu Aug 01 02:00:00 CEST 1996 until: Thu Dec 31 22:59:59 CET 2020
Certificate fingerprints:
         MD5:  5E:E0:0E:1D:17:B7:CA:A5:7D:36:D6:02:DF:4D:26:A4
         SHA1: 39:C6:9D:27:AF:DC:EB:47:D6:33:36:6A:B2:05:F1:47:A9:B4:DA:EA
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

6. Import the certificate and attach it to your server key pair by typing the command :

```
M:\keys>keytool -import -alias myserverkeypair -storepass YourPasswordHere \
                -keystore private.keystore -file myserver.cer
Certificate reply was installed in keystore
```

Key pair verification

Issue the following command :

```
M:\keys>keytool -list -v -alias myserverkeypair -storepass YourPasswordHere \
               -keystore private.keystore
Alias name: myserverkeypair
Creation date: Fri Aug 11 23:07:53 CEST 2000
Entry type: keyEntry
Certificate chain length: 2
Certificate[1]:
Owner: CN=www.myserver.com, OU=Foo Dept, O=Bar, L=Paris, ST=France, C=FR
Issuer: CN=Thawte Test CA Root, OU=TEST, O=Thawte, ST=FOR TESTING, C=ZA
Serial number: 69f12
Valid from: Fri Aug 11 23:00:07 CEST 2000 until: Mon Sep 11 23:00:07 CEST 2000
Certificate fingerprints:
         MD5:  41:84:55:8C:A1:85:28:DA:B0:5A:47:D6:5B:D2:ED:41
         SHA1: 61:DE:DB:E6:7C:3C:AD:90:63:9B:20:E0:FF:3B:02:3A:60:EB:B4:82
Certificate[2]:
Owner: CN=Thawte Test CA Root, OU=TEST, O=Thawte, ST=FOR TESTING, C=ZA
Issuer: CN=Thawte Test CA Root, OU=TEST, O=Thawte, ST=FOR TESTING, C=ZA
Serial number: 0
Valid from: Thu Aug 01 02:00:00 CEST 1996 until: Thu Dec 31 22:59:59 CET 2020
Certificate fingerprints:
         MD5:  5E:E0:0E:1D:17:B7:CA:A5:7D:36:D6:02:DF:4D:26:A4
         SHA1: 39:C6:9D:27:AF:DC:EB:47:D6:33:36:6A:B2:05:F1:47:A9:B4:DA:EA
```

As you can see the alias myserverkeypair points to a keyEntry type entry, its certificate chain has 2 certificate, the first being your own certificate, signed

by the Thawte Test CA Root, and the other being the Thawte Test CA Root own.**Resin configuration (resin.xml)**

add the support for the SunJSSE security provider :

```
<resin xmlns="http://caucho.com/ns/resin">
    <security-provider id='com.sun.net.ssl.internal.ssl.Provider'/>

<!-- declare a new HTTP server on port 443 (standard port for HTTPS),
      - with SSL enabled -->

<server>
  <!-- the http port -->
  <http port="80"/>

  <!-- the srun port, read by both JVM and plugin -->
  <cluster>
    <srun host='localhost' port='6802'/>
  </cluster>

  <http port=443>
    <jsse-ssl>
      <key-store-type>jks</key-store-type>
      <key-store-file>file://m:/keys/private.keystore</key-store-file>
      <password>YourPasswordHere</password>
    </jsse-ssl>
  </http>
```

Test !

Try connecting to your server with https instead of http !

I've been running successfully SSL on Resin with JDK 1.3 on Windows NT 4 SP6 and JDK 1.2.2 on Solaris 7.

And the fun begins when mixing HTTPS and WAP... !

Security Manager

In ISP environments, it's important that each user have restricted permissions to use the server. Normally, the web server will be run as a non-root user so the users can't read system files, but that user will still have read access. The use of RMI also requires a security manager.

Don't use a security manager if you're not in an ISP environment or using RMI. There's no need for it and the security manager does slow the server down somewhat.

Adding a Java security manager puts each web-app into a "sandbox" where Java limits the things that can be done from code within th web-app.

The security manager is enabled by adding a tag in the resin.xml.

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">

  <security-manager/>

  ...
```

enabling security-manager in resin.xml

java.policy

The security manager determines a `policy` that applies to the current virtual machine. The security manager is controlled by policy file's.

The simplest way to change the policy is to change one of the default policy file's. There are two default policy files that are used by the JDK:

```
${java.home}/lib/security/java.policy
${user.home}/.java.policy
```

An additional policy file can be set using the `java.security.policy` system property at the command line:

```
unix$ bin/resin.sh -Djava.security.policy=\textit{file:/path/to/java.policy}
win$ bin/resin.exe -Djava.security.policy=\textit{file:/path/to/java.policy}
```

The resulting policy for the virtual machine is the union of all granted permissions in all policy files.

java.policy syntax

A useful resource is Sun's documentation about security, in particular the policy permissions and policy file syntax files are useful.

Each web-app automatically has permissions to read, write and delete any file under the web-app's directory, including WEB-INF. It also has read permission for the classpath, including <classpath> from the <host> and <server> contexts.

```
#
# Permissions allowed for everyone.
#
grant {
  permission java.util.PropertyPermission "*", "read";
  permission java.lang.RuntimePermission "accessClassInPackage.*";
  permission java.net.SocketPermission "mysql.myhost.com:3306" "connect";
  permission java.io.FilePermission "/opt/resin/xsl/*", "read";
};

#
# Give the system and Resin classes all permissions
#
grant codeBase "file:${'${'}resin.home}/lib/-" {
        permission java.security.AllPermission;
};

grant codeBase "file:${'${'}java.home}/lib/-" {
        permission java.security.AllPermission;
};

grant codeBase "file:${'${'}java.home}/jre/lib/-" {
        permission java.security.AllPermission;
};

#
# Give a specific web-app additional permissions.
#
grant codeBase "file:/opt/web/webapps/ejb/WEB-INF/-" {
        permission java.io.FilePermission "/opt/web/doc/*", "read";
};
```

sample java.policy

Chapter 15

Inversion of Control

15.1 Resin IoC

See Also

- The Resin EJB page gives more information about the EJB bean lifecycles and their integration with Resin IoC.
- The Resin messaging page gives more information about the message bean and its integration with Resin IoC.
- The Resin remoting page shows Resin's remoting integration.
- All Resin configuration, including all JavaEE specified files uses Resin-IoC as the configuration engine.

Overview

Resin's IoC support is integrated with EJB 3.0 and the core components like Servlets, Filters and remote objects. This integration means plain Java beans can use EJB annotations and interception, EJBs can use Resin IoC annotations, and both kinds of beans can be configured directly from the `resin-web.xml` or discovered by classpath scanning.

So it's best to think of Resin-IoC as a set of orthogonal capabilities that are available to any registered bean. The basic capability types are:

- **Lifecycle model:** Java, `@Stateless` , `@Stateful` , or `@MessageDriven` . Resin-managed objects like Servlets and Filters are Java model beans.
- **Dependency injection:** injection annotations `@In` , `@Named` , `@EJB` , `@PersistenceUnit` , etc are available to all beans.
- **Registration:** all beans are registered in a unified typed-namespace registry (i.e. the registration half of dependency injection.)

- **Lifecycle events:** the `@PostConstruct` and `@PreDestroy`
- **Predefined aspects:** the `@TransactionAttribute` , `@RunAs` , `@RolesAllowed` , etc. annotations are available to all beans.
- **Custom interceptors:** EJB-style `@AroundInvoke` , and `@Interceptors` , as well as WebBeans-style `@Interceptor` and `@InterceptorBindingType` are available to all beans.
- **Event handling:** the WebBeans `@Observes` capability is available to all beans.

Injecting Resources

Before dependency injection, applications needed to use JNDI to grab resources managed by Resin: database connections, JMS queues, JCA `EntityManagers` , timers, `UserTransaction` , the JMX `MBeanServer` , etc. The JNDI lookup had two main flaws: it required a good chunk of boilerplate code to solve a simple problem, and it was untyped. Since JNDI is essentially a big HashMap, the only way of making sure your `DataSource` name didn't conflict with your JMS `Queue` was strictly structuring the JNDI names with patterns like `java:comp/env/jdbc/foo` .

With dependency injection, Resin will lookup, verify and inject the resource when it creates your managed class, for example when creating a servlet. In the following example, when Resin creates MyServlet, the `@In` annotation tells it to look for a `UserTransaction` and `DataSource` and use reflection to set the fields, before calling the servlet's `init()` method.

```
import javax.sql.DataSource;
import javax.transaction.UserTransaction;
import javax.webbeans.In;

public class MyServlet extends GenericServlet {
  @In private DataSource _ds;
  @In private UserTransaction _ut;

  ...
}
```

Example: DataSource and UserTransaction

@Named and bindings

Since many applications use multiple resources like named databases, your injection may need to specify a name or other binding to uniquely identify the resource you want. In the case of a unique resource like `UserTransaction` or `MBeanServer` or an application with a single `DataSource` , the `@In` is enough information. When you have multiple databases, you'll want to use `@Named` to specify the database name.

```
import javax.sql.DataSource;
import javax.webbeans.Named;

public class MyServlet extends GenericServlet {
  @Named("foo") private DataSource _ds;

  ...
}
```

Example: @Named with DataSource

```
<web-app xmlns="http://caucho.com/ns/resin">

  <database>
    <name>foo</name>
    <driver type="org.gjt.mm.mysql.Driver">
      <url>jdbc:mysql://localhost:3306/foo</url>
    </driver>
  </database>

  <database>
    <name>bar</name>
    <driver type="org.gjt.mm.mysql.Driver">
      <url>jdbc:mysql://localhost:3306/bar</url>
    </driver>
  </database>

</web-app>
```

Example: resin-web.xml database configuration

Injection using a `@Named` binding annotation is still typed. You can have a database with `@Named("foo")` and a JMS queue with `@Named("foo")` , i.e. each type has its own namespace. The typed injection is a big improvement from the JNDI `java:comp/env/jdbc/foo` vs `java:comp/env/jms/foo` conventions.

Although many applications will just use `@In` and `@Named` , you can also create your own `BindingType` annotations to match resources and components.

field, method, and constructor injection

Since Resin implements all three flavors of dependency injection: field, method and constructor, the choice of style is up to you.

You can mark any field with `@In` , `@New` , `@Named` or any other `@BindingType` to tell Resin to inject the field. When a `@BindingType` is used, Resin will only match components configured with that binding type.

```
import javax.webbeans.Named;

public class MyBean {
  @Named DataSource _ds;
  ...
}
```

Example: field injection with @Named

Method injection can use any binding annotation on any of the parameters. When Resin introspects the component class and it finds any `@BindingType` or `@In` parameter on a method, it will schedule that method to be injected. Method parameters can also use `@New` . The method does not need to follow bean-style setting conventions; any method will work.

```
import javax.webbeans.Named;

public class MyBean {
  void foo(@Named("jdbc/foo") DataSource myDataSource) { ... }
  ...
}
```

Example: method injection with @Named

Construction injection is available for components and singleton beans. Like method injection, the `@BindingType` values like `@Named` assigned to the parameters determine the values to be injected.

If the bean has multiple constructors, exactly one must be marked `@In` or have a `@BindingType` value.

```
import javax.webbeans.In;

public class MyBean {
  @In
  public MyBean(DataSource myDataSource) { ... }
  ...
}
```

Example: constructor injection

WebBeans-enabled components (injectable objects)

Any Resin-managed object can use the entire WebBeans dependency-injection system and all of the managed objects, while objects you create using `new` are still plain Java objects. Once you've got a root object managed by the system, any futher WebBeans components or singletons you bring in will also be managed. The starting set of managed objects is pretty broad and includes:

- <bean> (singletons) defined in the resin.xml or resin-web.xml
- <component> (WebBeans components) defined in the resin.xml, resin-web.xml or web-beans.xml
- EJB message-driven beans
- EJB session beans
- Filters (servlet)
- JSF resources (currently only model beans)
- JSP pages
- JSP tag libraries
- <resource> components (Resin will be migrating to use <bean> for resources in the future.)
- Servlets
- Servlet Listeners defined in the web.xml
- WebBeans annotated components (`@Component`) discovered through the classpath scan (see below)
- WebBeans components injected with `@New` (see below)
- Remote services defined by <servlet> (e.g. Hessian or SOAP).

Resin global resources

Resin automatically provides several global resources to any injectable code. Since these resources are unique, your application will use `@In` as the annotation.

`javax.webbeans.Container` provides a reference to the WebBeans container itself. Applications can use the `Container` to raise `WebBeans` events, and lookup components programmatically.

`javax.webbeans.Conversation` injects the WebBeans conversation scope for a JSF application. The conversatio scope lets JSF views store data local to the page itself, separate from the HTTP session scope.

`javax.management.MBeanServer` injects the JMX management server. JMX manages Resin's resources, so an application can use JMX to view the current state of Resin. See the JavaDoc for an overview of Resin's resources.

`javax.transaction.TransactionManager` injects the XA transaction manager. Advanced applications which want to add their own `XAResource` or `Synchronization` objects to an active transaction can use `@In TransactionManager` .

`javax.transaction.UserTransaction` injects the XA user transaction manager. Applications controlling XA transactions programmatically will use

the `UserTransaction` to `begin()` , `rollback()` , and `commit()` distributed transactions.

`java.util.concurrent.ScheduledExecutorService` lets applications schedule threads and timed events controlled by Resin's thread pool, and restricted by the thread-max configuration in the resin.xml. This `ScheduleExecutorService` is classloader-safe, so Resin will automatically close your scheduled tasks when restarting a web-app.

Resin configured resources

All Resin configured resources are available through WebBeans, since Resin uses WebBeans as its internal registry.

<`bean`> singletons are application-specific custom services. They are exactly like the <component> beans but default to singleton scope. Like <component> they can either use WebBeans annotations or just be POJO objects. The <bean> tag makes the beans available to any other WebBeans component. The bean default to `@Singleton` scope, which means that a unique instance is created at Resin start time. The injected type will depend on the `class` attribute of the bean.

<`component`> objects are application-specific custom beans. They can either use WebBeans annotations or just be POJO objects. The <component> tag makes the beans available to any other WebBeans component. The components default to `@Dependent` scope, which means a new instance is created each time they're referenced or injected. The injected type will depend on the `class` attribute of the bean.

<`database`> is a JDBC pooled database. The `name` attribute is assigned to the `@Named` binding. Databases are injected as `javax.sql.DataSource` types.

EJB stateless beans are automatically registered the WebBeans. The default `@Named` value is the ejb-name. Stateless beans can use the <ejb-stateless-bean> tag for custom configuration, or rely on the standard EJB discovery mechanism. Each EJB 3.0 `@Local` interface is registered separately in the WebBeans registry.

EJB stateful beans are automatically registered with WebBeans. The default `@Named` value is the ejb-name. Stateful beans can use the <ejb-stateful-bean> tag for custom configuration, or rely on the standard EJB discovery mechanism. Each injection or bean lookup will return a new stateful bean reference, modified by the bean's scope (see below.) Each EJB 3.0 `@Local` interface is registered separately in the WebBeans registry.

EntityManager and **EntityManagerFactory** objects from JPA are automatically registered with WebBeans. The default `@Named` value is the `persistence-unit` name.

<**env-entry**> tags register their values with WebBeans. The default `@Named` value is the `env-entry-name` . The registered type is the `env-entry-type` .

<**jms-connection-factory**> automatically registers the configured factory with WebBeans. Connection factories can also be configured with <bean> or

<resource>, depending on the JMS provider. The registered type is the `class` of the connection factory, usually `javax.jms.ConnectionFactory`

<**jms-topic**> automatically registers the configured topic with WebBeans. The default `@Named` value is the topic name. Topic can also be configured with <bean> or <resource>, depending on the JMS provider. The registered type is `javax.jms.Topic`

<**jms-queue**> automatically registers the configured queue with WebBeans. The default `@Named` value is the queue name. Queue can also be configured with <bean> or <resource>, depending on the JMS provider. The registered type is `javax.jms.Queue`

<**remote-client**> registers remoting clients with WebBeans. The <remote-client> tag will configure the protocol used and the expected proxy class. The registered type is the API `class` of the remote service.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <remote-client class="example.HelloService">
    <url>hessian:http://localhost:8080/hello/</url>
  </remote-client>

</web-app>
```

Example: resin-web.xml for remote-client

```
public class MyServlet extends GenericServlet {
  @In example.HelloService _hello;
  ...
}
```

Example: Injecting for remote-client

Application components

The primary value of Resin's dependency injection system is as a type-safe component and service organization registry. Module and component-oriented software has been a goal of software developers for decades, but in practice developing components and services is a difficult task, in large part because the configuration and assembly of the components has been just as complicated as the code itself. There are no silver bullets in software, but the WebBeans registry does significatly reduce the housekeeping code and XML which gets in the way of good design.

Annotation-based Component Discovery

At startup, Resin will scan jars and class directories for a `META-INF/web-beans.xml` file. The presence of that file tells Resin to

start scanning all the classes in the jar or directory for `@Component` annotations. Resin will register each class with a `@Component` market with the WebBeans directory. The `META-INF/web-beans.xml` can be trivial, since its primary purpose is speeding up startup time by letting Resin scan only the jars which contain components. So, applications will generally want to minimize the number of classes in `@Component` jars to make startup fast.

```
<web-beans xmlns="http://caucho.com/ns/resin">
</web-beans>
```

Example: META-INF/web-beans.xml

The component itself can be any Java class. Since Resin manages the component, it can use `@In` and `@Named` to inject any other component or resource. Resin will take care of any circular dependencies automatically.

```
import javax.annotation.PostConstruct;
import javax.webbeans.Component;

@Component
public class Movie {
  private String _title;
  private String _director;

  public String getTitle() { return _title; }
  public void setTitle(String title) { _title = _title; }

  public String getDirector() { return _director; }
  public void setDirector(String director) { _director = director; }

  @PostConstruct
  public void init()
  {
    ...
  }
}
```

Example: Movie.java

The `@PostConstruct` annotation tells Resin to call the `init()` method once all the injection and configuration is complete.

Any other component or Resin-managed class like servlets can now use `@In Movie` to inject the movie resource.

```
import javax.webbeans.In;

public class MyServlet extends GenericServlet {
  @In Movie _movie;

  ...
}
```

Example: MyServlet.java

Most application components will use the `@Component` discovery mechanism. Except for the `META-INF/web-beans.xml` marker file, no additional housekeeping code or XML is required. As described below, applications can reduce the configuration by one more step with the `@New` annotation replacing `@In` . The target class of the `@New` annotation is automatically registered with WebBeans even if the class has no `@Component` annotation or if the `META-INF/web-beans.xml` is missing. In other words, any plain Java class can become a part of the WebBeans system without overhead.

Scopes: @Singleton, @Dependent, @RequestScoped

The scope of the component determines when Resin will create a new component and when it will reuse an old one. Singletons will always return the same object, while dependent components will always return a new object instance. Long-lived services will typically be singletons, while scratch-space modules will be dependent components.

Components default to `@Dependent` scope. Since many components will be used as pieces of other components, `@Dependent` is the least-surprising value as a default.

You can specify a component's scope with an annotation or in the <`bean`> or <`component`> tag. The predefined scope values are: `@Dependent` , `@RequestScoped` , `@ConversationScoped` , `@SessionScoped` , `@ApplicationScoped` and `@Singleton` .

- `@Dependent` creates a new instance each time.
- `@RequestScoped` creates a new instance for each servlet request, reusing the instance for the same request.
- `@ConversationScoped` creates a new instance for each JSF conversation, i.e. for each JSF view page.
- `@SessionScoped` creates a new instance for each HTTP session, reusing the instance for the same session.
- `@ApplicationScoped` uses a single instance in each web-app. For web applications, this will have the same lifecycle as `@Singleton` .

- `@Singleton` uses a single instance in the container. For web applications, this will have the same lifecycle as `@ApplicationScoped` .

An example scoped resource might be a `Calculator` object which is used only for a single instance. It might fill the arguments while processing a form and then calculate the result. The `@RequestScoped` makes sure scripts receive the same instance each time it's called.

```
import javax.webbeans.RequestScoped;
import javax.webbeans.Component;

@RequestScoped
@Component
public class Calculator {
  private int _a;
  private int _b;

  public void setA(int a) { _a = a; }
  public void setB(int b) { _b = b; }

  public int getSum() { return _a + _b; }
}
```

Example: @RequestScoped Calculator

You could also register the same calculator using XML:

```
<web-app xmlns="http://caucho.com/ns/resin">

  <component class="example.Calculator" scope="request"/>

</web-app>
```

Example: resin-web.xml Calculator

@New Component Discovery

The `@New` annotation automatically registers the target class with the WebBeans registry and tells Resin to create a new instance of the bean, even if the bean has a singleton scope definition. Since `@New` replaces `@In` , it can be used anywhere `@In` can be used.

```
import javax.webbeans.New;

public class MyServlet extends GenericServlet {
  @New Movie _movie;

  ...
}
```

Example: MyServlet.java

The `Movie` is identical to the movie class defined above, but doesn't need the `@Component` annotation and doesn't need the `META-INF/web-beans.xml` marker file. In many cases, the `@New` annotation can replace the Java `new` operator. If your application starts using `@New` consistently, it can add injection capabilities to a growing share of your code, letting you simplify and refactor incrementally.

```
public class Movie {
  private String _title;

  public String getTitle() { return _title; }
  public void setTitle(String title) { _title = title; }

  @PostConstruct
  public void init() { ... }
}
```

Example: Movie.java

Lifecycle: @PostConstruct and @PreDestroy

If your service needs to initialize itself after being configured, it can annotation an initialization method with `@PostConstruct` . After Resin creates, injects, and configures your component, it will call any `@PostConstruct` methods. Long-lived services or services that need to register themselves with other services will typically need to use the `@PostConstruct` annotation.

At the end of a component's lifetime, you might need to close some resources, e.g. closing a socket or delisting from a timer service. Resin will call any component method marked with `@PreDestroy` before it destroys the method.

For example, a `TimerService` may want to schedule a event every 2 seconds. The `@PostConstruct` method will start the timer and the `@PreDestroy` method will stop the timer.

```
import javax.annotation.PostConstruct;
import javax.webbeans.In;
import javax.webbeans.Component;
import java.util.concurrent.ScheduledExecutorService;
import com.caucho.webbeans.Singleton;

@Component
@Singleton
public class TimerService {
  @In ScheduledExecutorService _timer;

  @PostConstruct
  public void init()
  {
    _timerFuture = _timer.scheduleAtFixedRate(this, 0, 2, TimeUnit.SECONDS);
  }

  ...

  @PreDestroy
  public void close()
  {
    _timerFuture.cancel(false);
  }
}
```

Example: Timer Service

XML configuration

You can register your components and services with Resin using the resin.xml or resin-web.xml files. Since the WebBeans registry is integrated with Resin, your services be treated as first-class components along with the Resin resources. Although most components will not need XML, there are a few advantages for the small number of services which do use XML.

The XML-configuration lets you customize your application for a particular environment, e.g. setting configuration parameters. For example, Resin's <database> needs to select a database driver and configure the URL, user and password of the database as well as configuring connection pooling parameters. Some application services will also need configuration.

In addition, the XML-configuration documents the services you've enabled. For heavyweight services, this documentation is critical, while lightweight components do not need this extra housekeeping overhead.

bean and component registration

The <bean> and <component> tags register application classes with Resin. The two tags are identical except for the expected lifecycle: <bean> configures singleton services while <component> configures component template classes.

In other words, the default scope of a <bean> is @Singleton while the default scope of a <component> is @Dependent . A <component> will create a new instance each time it's injected or referenced, while a <bean> always returns the same singleton instance.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <bean class="example.MyService"/>

  <component class="example.MyService"/>

</web-app>
```

Example: bean and component resin-web.xml

The <bean> and <component> tags have a number of optional attributes:

ATTRIBUTE	DESCRIPTION
binding	the @BindingType annotations for custom bean matching
class	the Java class of the bean or component (required)
init	optional configuration/initialization block, using bean-style assignment
name	the instance name, used for @Named and script integration
scope	specifies scope of the instances: request, conversation, session, application, or singleton

bean and component attributes

Bean property configuration

Resin's XML configuration uses the standard JavaBeans patterns to configure properties. Resin uses the same mechanism for all of its own configuration parsing, including every JavaEE configuration file, the resin-web.xml and the resin.xml itself. So your application will have all the configuration flexibility it needs.

Since the component beans can use WebBeans injections, injected components are typically not configured in the resin-web.conf, avoiding the need for tags like <ref> .

```
public class Hello {
  private String _greeting = "default";

  public void setGreeting(String greeting) { _greeting = greeting; }
  public String getGreeting() { return _greeting; }
}
```

Example: Hello.java

The basic example sets a `greeting` property of a hello, world bean. In this example, we're configuring a singleton, but the <`init`> works for dependent components as well. Resin will apply the configuration to the instance as part of the creation process.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <bean class="example.Hello">
    <init>
      <greeting>Hello, World</greeting>
    </init>
  </bean>

</web-app>
```

Example: resin-web.xml configuring a singleton

Resin's configuration uses 5 basic bean patterns, extending the JavaBeans conventions. It can configure literal values like string and integers as well as configuring other beans. Any component bean configured by Resin has full access to `@In` injection as well as the standard `@PostConstruct` annotations. Sub-beans are not automatically registered with WebBeans, i.e. they act like the servlet configuration.

(Currently the patterns are name-based like JavaBeans, since Resin was designed before annotations. We may add configuration annotations in the future.

```
  public void setFoo(String data);

  public void setFoo(Movie data);

  public void addFoo(Movie data);

  public Movie createFoo();

  public void setText(String data);
```

Example: Bean configuration patterns

1. `setFoo(String)` configures a standard JavaBeans-style literal.

2. `setFoo(Movie)` creates a new instance of `Movie` and recursively configures it.

3. `addFoo(Movie)` also creates a new instance of `Movie` and recursively configures it. `addFoo` is an easy way of configuring lists.

4. `Movie createFoo()` lets the bean create the `Movie` instance. Many beans can use `createFoo` with inner classes to handle complex configuration.

5. `setText` is called with the text contents of the XML. Value-style beans will use this. (somewhat rare).

As mentioned above, Resin uses these 5 patterns to handle all of the JavaEE configuration files. In particular, the `createFoo` pattern returning inner classes is very handy for some complicated configuration cases, and for cases where a sub-tag needs information about the parent.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <bean class="example.Theater">
    <init>
      <name>Balboa</name>

      <movie title="The Princess Bride"/>

      <movie title="The Maltese Falcon"/>
    </init>
  </bean>

</web-app>
```

Example: sub-bean configuration example

In this example, the `Theater` classes uses an inner `Movie` class to illustrate the use of the `create` pattern.

```
public class Theater {
  String _name;

  ArrayList<Movie> _movies = new ArrayList<Movie>();

  public void setName(String name) { _name = name; }

  public Movie createMovie()
  {
    return new Movie(this);
  }

  public void addMovie(Movie movie)
  {
    _movies.add(movie);
  }

  public static class Movie {
    private Theater _theater;
    private String _title;

    Movie(Theater theater)
    {
      _theater = theater;
    }

    public void setTitle(String title) { _title = title; }
  }
}
```

Example: Theater.java

Base configuration: string conversions

Resin-IoC provides a number of built-in string conversion types as well as supporting JavaBeans `PropertyEditor` and custom converters.

TYPE	DESCRIPTION
boolean, Boolean	Java boolean
byte, Byte	Java byte
short, Short	Java short
int, Integer	Java integer
long, Long	Java long
float, Float	Java float
double, Double	Java double
char, Character	Java char
String[]	String array separated by commas
Class	Java classes
Path	Resin VFS Paths
File	java.io.File

URL	java.net.URL
Pattern	java.util.regex.Pattern
Locale	java.util.Locale
Date	java.util.Date
Properties	java.util.Properties
RawString	com.caucho.config.type.RawString

Built-in String Converters

enumerations Enumerations are automatically converted from their string representation.

String constructor Resin-IoC will automatically convert a string to an object if the object has a single String argument constructor.

```
public class MyBean {
  public MyBean(String value)
  {
    ...
  }
}
```

Example: MyBean with constructor

valueOf For classes which implement a static `valueOf(String)` method, Resin will automatically convert to the given type using the `valueOf` method.

```
public class MyBean {
  ...

  public static MyBean valueOf(String text)
  {
    MyBean bean = new MyBean();
    bean.setTextValue(text);
    bean.init();
    return bean;
  }
}
```

Example: MyBean with valueOf

setValue For objects with a `setValue` or `addText` method and a zero-argument constructor, Resin-IoC will convert using the following steps:

1. Create the object
2. Inject any dependencies

3. Call `setValue` or `setText` with the string
4. Call any @PostConstruct
5. Return the configured bean

Compound types

list Setters taking a `List` or array argument can be configured with list values.

List items are specified directly with <value> elements. There is no extra <list> element required. The <list> element is only used when creating a sub-list or sub-element (see below.)

```
<my-bean>
  <values>
    <value>a</value>
    <value>b</value>
    <value>c</value>
  </values>
</my-bean>
```

Example: MyBean.setValues(List)

```
<my-bean>
  <values>
    <value>a</value>
    <value>b</value>
    <value>c</value>
  </values>
</my-bean>
```

Example: MyBean.setValues(String [])

In the following example, the argument is an object, so we need a <list> element to tell Resin to create a list. The object created will be an `ArrayList` .

```
<my-bean>
  <values>
    <list>
      <value>a</value>
      <value>b</value>
      <value>c</value>
    </list>
  </values>
</my-bean>
```

Example: MyBean.setValues(Object)

Resin-IoC can always use the `addXXX` pattern to add a variable number of items. Normally, the `addXXX` pattern is easier and more maintainable than the

`addList` pattern. In particular, validation of the item values is quicker and more accurate with `addXXX` .

```
<my-bean>
  <value>a</value>
  <value>b</value>
  <value>c</value>
</my-bean>
```

Example: MyBean.addValue(String)

map Generic maps can use an <entry> syntax to define property values.

```
<my-bean>
  <values>
    <entry key="a" value="one"/>
    <entry key="b" value="two"/>
    <entry key="c" value="three"/>
  </values>
</my-bean>
```

Example: MyBean.setValues(Map)

References and EL Expressions

Resin-IoC configuration files can use EL expressions to get references to resources, beans, system properties, and calculate generatal expressions based on those values. Since all Resin's resources are added to the WebBeans registry automatically, application components have access to anything they need.

Both the JSP immediate syntax and deferred syntax are supported (${...} vs #{...}). Currently, there is no distinction between the two, but the deferred syntax is preferred, since Resin-IoC initializes beans lazily to handle circular references.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <bean name="a" class="qa.FooBean">
    <init bar="#{b}"/>
  </bean>

  <bean name="b" class="qa.BarBean">
    <init foo="#{a}"/>
  </bean>

</web-app>
```

Example: circular references in resin-web.xml

Because Resin's EL implementation allows method expressions, you can use beans as factories in the EL expressions.

Scripting: PHP, JSF and JSP

If your application is using a scripting language like PHP for the presentation layer, the WebBeans registry provides a simple interface to use your components and services. Although WebBeans injection is typed, each WebBean is also registered under its name for scripting applications. The name must be globally unique, of course, unlike the typed injection binding.

The default scripting name is the name of the component's class, with the first character lowercased. You can change the name by adding a `@Named` or setting a `name` attribute in the <`component`> or <`bean`> .

```
@Component
@RequestScoped
public class Movie {
  ...
}
```

Example: Movie.java

Quercus/PHP

In Quercus/PHP, the `java_bean(name)` method returns the component with the given name. For singletons, Resin will return the unique bean. For other beans, Resin will create the bean if necessary and store it in the configured scope.

```
<?php

$movie = java_bean("movie");

echo "title: " . $movie->title . "\n";

?>
```

Example: accessing the movie from PHP

JSP/JSF EL

Resin automatically provides the WebBeans variables to EL expressions for both JSP and JSF.

```
<c:out value="${movie.title}"/>
```

Example: accessing the movie from JSP

The JSF also uses the expression language, but uses the deferred syntax, since JSF needs to build a component tree first.

```
<f:view>
  <h:outputText value="#{movie.title}"/>
</f:view>
```

Example: accessing the movie from JSF

@BindingType: custom injection binding

Although many applications will just use `@In` and `@Named` injection binding, applications can create their own binding annotations. Some bindings might be more logical, for example an `@XA` annotation might mark the transactional `DataSource` , while `@ReadOnly` might mark a read-only `DataSource` . Drivers might use `Scheme("mysql")` to allow for URL-based configuration, like the "jdbc:mysql://localhost:3306/test" of JDBC.

```
import com.foo.webbeans.ReadOnly;
import com.foo.webbeans.XA;

public class MyServlet extends GenericServlet {
  @ReadOnly DataSource _readDatabase;
  @XA DataSource _xaDatabase;

  ...
}
```

Example: ReadOnly and XA databases

You can create a custom binding annotation using the `@BindingType` annotation. When Resin introspects the injection point and the components, it will look for any annotation with the `@BindingType` meta-annotation.

The annotation can also have annotation parameters. If they exist, Resin will make sure only matching components will be injected.

The custom binding annotation can be used anywhere predefined binding annotations can, including fields, methods, constructor, producers, or event observers.

```
package com.foo.webbeans;

@BindingType
@Target({TYPE, METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface ReadOnly {
}
```

Example: ReadOnly.java

@Produces methods

Some components are more easily produced using a factory method rather than getting instantiated directly. In those cases, your application can mark a factory component's method with the `@Produces` annotation. Resin will register the results of the method with WebBeans.

```
import javax.webbeans.Component;
import javax.webbeans.Produces;
import com.caucho.webbeans.Singleton;

@Component
@Singleton
public class MyFactory {
  @Produces public List<Movie> currentMovies()
  {
    ...
  }
```

Example: @Produces

The produces method can be marked with `@ScopeType` , `@ComponentType` or `@BindingType` annotations just like a class-based component can.

Aspects: Method Interception

Some functions like security, logging and transaction handing need to be used for each method invidually, but require a common implementation pattern. The WebBeans AOP uses a single method as a interceptor for each method invocation.

Your method will use an `@InterceptorType` annotation letting Resin know where it should apply the interceptor:

```
import com.foo.webbeans.Secure;

@Component
public class MyBean {
  @Secure
  public void doSomethingSafely() { ... }
}
```

Example: secure method

The implementation class will use the `javax.interceptor.*` API to implement the interceptor.

```
import javax.interceptor.*;
import javax.webbeans.Interceptor;

@Secure @Interceptor
public class MySecureInterceptor {
  @AroundInvoke
  public Object checkSecurity(InvocationContext inv) throws Exception
  {
    if (! myContextIsSecure())
      throw new MySecurityException("permissiong denied");

    return inv.proceed();
  }
}
```

Example: security implementation

The interceptors must be enabled in the `META-INF/web-beans.xml` file. This protects your code from any surprising interceptors.

```
<web-beans xmlns="http://caucho.com/ns/resin">
  <interceptors>
    <interceptor>com.foo.MySecureInterceptor</interceptor>
  </interceptors>
</web-beans>
```

Example: META-INF/web-beans.xml

Event Handling

Your components can also handle events thrown through the WebBeans API. Any method with an `@Observes` parameter will receive events with the proper type. The event can be any Java class.

```
import javax.webbeans.Component;
import javax.webbeans.Observes;

@Component
public class MyHandler {
  public void myHandlerMethod(@Observes Movie movie)
  {
    ...
  }
}
```

Example: event handler

Your application can throw events through the WebBeans `Container` API, which is available through injection:

```
import javax.webbeans.Container;

public void MyServlet extends GenericServlet {
  @In Container _webbeans;

  public void service(ServletRequest req, ServletResponse res)
  {
    _webbeans.raiseEvent(new Movie("A Bridge Too Far"));
  }
}
```

Example: raising events

For the above example, Resin will look for all components which have an `@Observes` method receiving a `Movie` and deliver the event to that component.

Aspect Annotations

@AroundInvoke

`@AroundInvoke` marks an interception method on the bean or an interceptor class. The interceptor is invoked while processing a business method.

```
@Target(METHOD)
@Retention(RUNTIME)
public @interface AroundInvoke {
}
```

```
import javax.interceptor.*;

public class MyBean {
  @AroundInvoke
  protected Object log(InvocationContext cxt)
    throws Exception
  {
    System.out.println("Before: " + cxt.getMethod());

    Object value = cxt.proceed();

    System.out.println("After: " + cxt.getMethod());

    return value;
  }

  public String hello()
  {
    return "hello, world";
  }
}
```

Example: @AroundInvoke method

@DenyAll

`@DenyAll` annotation marks a method as forbidden to all users.

```
@Target({METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface DenyAll {
}
```

@Interceptor

`@Interceptor` marks a class as an interceptor using WebBeans-style interception. The class will normally also have an `@AroundInvoke` method as well as any `InterceptorBindingType` annotations.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface Interceptor {
}
```

@InterceptorBindingType

`@InterceptorBindingType` is a WebBeans meta-annotation for creating interceptor binding types. Applications will use `@InterceptorBindingType` to create application-specific interceptors. The WebBeans-style of interception decouples the interception declaration from the interceptor classes, in contrast with the EJB-style which specifies the interceptor class directly.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface InterceptorBindingType {
}
```

@Interceptors

`@Interceptors` marks an method or class as being intercepted by the named classes. The interceptor classes will implement an `@AroundInvoke` method to process the `InvocationContext` .

VALUE	MEANING	DEFAULT
value	Lists the interceptor classes to apply to the method.	

@Interceptors properties

```
@Target({TYPE,METHOD})
@Retention(RUNTIME)
public @interface Interceptors {
  public Class []value();
}
```

```
import javax.interceptor.*;

public class MyBean {
  @Interceptors(MyInterceptor.class)
  public String hello()
  {
    return "hello, world";
  }
}

public class MyInterceptor {
  @AroundInvoke
  protected Object log(InvocationContext cxt)
    throws Exception
  {
    System.out.println("Before: " + cxt.getMethod());

    Object value = cxt.proceed();

    System.out.println("After: " + cxt.getMethod());

    return value;
  }
}
```

Example: @Interceptor method

InvocationContext

The `InvocationContext` API is used by invocation methods to examine the calling context, and possibly set parameters. A no-op interceptor would just call the `proceed()` method.

METHOD	DESCRIPTION
getContextData	Returns a map containing any context information
getMethod	Returns the called API method
getParameters	Returns the Java parameters for the call
getTarget	Returns the target object, i.e. the Java object that will receive the call after all the interceptors complete.

proceed	Call the next interceptor in the chain, or call the final object at the end of the chain.
setParameters	Sets the Java parameters for the call

InvocationContext methods

```
public interface InvocationContext {
  public Object proceed() throws Exception;

  public Map<String, Object> getContextData();
  public Method getMethod();
  public Object[] getParameters() throws IllegalStateException;
  public void setParameters(Object[] parameters) throws IllegalStateException;
  public Object getTarget();
}
```

@PermitAll

`@PermitAll` annotation marks a method as allowed for all users.

```
@Target({METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface PermitAll {
}
```

@RolesAllowed

`@RolesAllowed` lists all the roles (i.e. permissions) allowed to access the method. If the user in the security context does not match the role, an exception will be thrown.

VALUE	MEANING	DEFAULT
value	Lists the roles (permissions) that are allowed.	

RolesAllowed properties

```
@Target({TYPE,METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface RolesAllowed {
  String []value();
}
```

@RunAs

`@RunAs` changes the security context user to a defined role. Security tests within the context of the `@RunAs` will match the specified role.

VALUE	MEANING	DEFAULT
value	The role name to run as.	

RunAs properties

```
@Target({TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface RunAs {
  String value();
}
```

@TransactionAttribute

Defines the transaction boundary for business methods. The default value is REQUIRED. If @TransactionAttribute annotates the class, it defines the default value.

All Resin-managed beans can use `@TransactionAttribute` : @Stateful, @Stateless, @MessageDriven and plain Java beans.

VALUE	MEANING
REQUIRED	Start a new transaction if necessary
SUPPORTS	Don't start a new transaction, but use one if it exists
MANDATORY	Require the caller to have started a transaction
NEVER	Forbid the caller to have started a transaction
REQUIRESNEW	Always start a new transaction, suspending the old one
NOTSUPPORTED	Suspend any active transaction

TransactionAttributeType

- SUPPORTS is typically used for read-only methods
- REQUIRED is typically used for updating (read/write) methods

```
@Target({TYPE,METHOD})
@Retention(RUNTIME)
public @interface TransactionAttribute {
  TransactionAttributeType value() default REQUIRED;
}
```

Dependency Injection Annotations

@BindingType

`@BindingType` is a meta-annotation for creating custom injection binding types. Applications can create their own injection annotations like `@Named` to provide application-specific binding. For example, a database might have a `@XA` and a `@NonXA` connector implemented.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface BindingType {
}
```

```
package demo;

@BindingType
@Target({TYPE, METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface XA {
}
```

Example: custom @XA binding type

```
package demo;

import javax.sql.*;

public class MyBean {
  @XA DataSource _xa;
  @NonXA DataSource _nonXa;

  ...
}
```

Example: using the custom @XA binding type

@EJB

Configures EJB values for a field or method.

@EJB is essentially a @Resource where it's known that the result is an EJB interface.

PROPERTY	DESCRIPTION	DEFAULT
businessInterface	The EJB interface to lookup	The field type
name	The EJB name of the resource	The field name
jndiName	The jndi name of the resource	The EJB name

```
@Target({TYPE, METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface EJB {
  String name() default "";
  String businessInterface() default "";
  String jndiName() default "";
}
```

In the following exaple, Resin will call `setFoo` method with the bean in "java:comp/env/ejb/foo" before the session is started.

```
@EJB
void setFoo(example.Test test)
{
  _test = test;
}
```

Example: @EJB method injection

@In

Marks a field, method, or parameter for injection. When used with a constructor, it marks the given constructor as the constructor to use when creating new instances.

```
@Target({CONSTRUCTOR, METHOD, FIELD, PARAMETER, TYPE})
@Retention(RUNTIME)
public @interface In {
}
```

@Named

`@Named` is a predefined `@BindingType` annotation for named objects. Many application objects can just use `@Named` to disambiguate multiple objects with the same type.

PROPERTY	DESCRIPTION	DEFAULT
value	The name binding for the object to inject	required

@Named parameters

```
@BindingType
@Target({METHOD, FIELD, PARAMETER, TYPE})
@Retention(RUNTIME)
public @interface Named {
  String value();
}
```

@New

Marks a field, method, or parameter for injection with a new instance of the object. If the type of the `@New` has not yet been registered with the WebBeans directory, it will be registered automatically..

```
@Target({METHOD, FIELD, PARAMETER, TYPE})
@Retention(RUNTIME)
public @interface New {
}
```

@NonBinding

`@NonBinding` is a meta-annotation used for creating `@BindingType` annotations. It excludes an annotation property from the binding algorithm. Normally, the injection binding must matches all the propeties in the object's annotations with the properties in the injection property. The `@NonBinding` annotation skips the check for the annotated property.

```
@Target({FIELD, METHOD})
@Retention(RUNTIME)
public @interface NonBinding {
}
```

```
package demo;

@BindingType
@Target({TYPE, METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Demo {
  @NonBinding String description() default "";
}
```

Example: @Demo with @NonBinding

@Resource

`@Resource` provides JNDI-based resource injection. `@Resource` can also be used at the Class level to declare a dependency in cases where the session bean loads the JNDI value by itself.

In general, it's better to use the WebBeans annotations: `@In` , `@Named` or custom `@BindingType` annotations, since they use the type-safe WebBeans registry instead of JNDI. `@Resource` is supported for backwards compatibility.

PROPERTY	DESCRIPTION	DEFAULT
authenticationType	What kind of authentication is expected for the resource: APPLICATION or CONTAINER	CONTAINER
description	An optional description of the resource	
name	The jndi-name of the resource	java:comp/env/ `class-name` # `field-name`
type	The class of the expected resource	The field type
shareable	True if the bean follows JCA shareability requirements.	true
mappedName	The produce-specific name of the resource	The field name

Framework Integration

Frameworks like Struts2, Wicket, and Mule can delegate object creation to Resin-IoC. By configuration Resin-IoC to create the framework objects, your application's components can directly inject Resin resources and application components configured with Resin.

Integration information for the frameworks is maintained on the Caucho wiki site. Currently supported frameworks include:

- http://wiki.caucho.com/Mule
- http://wiki.caucho.com/Spring
- http://wiki.caucho.com/Struts2
- http://wiki.caucho.com/Wicket

Implementing new object factories for other frameworks is straightforward.

ObjectFactory pattern

Frameworks like Struts2 provide an `ObjectFactory` pattern. In this case, the framework gives a `Class` object and asks the factory to create a new instance. The `ObjectFactory` is the most powerful pattern, since objects can use @Observes, @InterceptionType, @TransactionAttribute and even @Stateless, as well as the usual dependency injection.

```
package com.caucho.xwork2;

import com.caucho.webbeans.manager.*;
import com.opensymphony.xwork2.ObjectFactory;
import java.util.*;
import javax.webbeans.*;

public class ResinObjectFactory extends ObjectFactory
{
  private final WebBeansContainer _webBeans = WebBeansContainer.create();

  @Override
  public Object buildBean(Class clazz, Map extraContext)
    throws Exception
  {
    return _webBeans.getObject(clazz);
  }
}
```

Example: Struts2 integration

Injection pattern

Some frameworks, like Wicket, prefer to instantiate the object, but can call Resin-IoC for a dependency-injection step. With this kind of framework inte-

gration, the created object only gets dependency-injection; it does not get any aspects or interception.

```
package com.caucho.wicket;

import com.caucho.webbeans.manager.*;

import org.apache.wicket.Component;
import org.apache.wicket.application.*;

public class ResinComponentInjector implements IComponentInstantiationListener
{
  private WebBeansContainer _webBeans = WebBeansContainer.create();

  public void onInstantiation(Component component)
  {
    _webBeans.injectObject(component);
  }
}
```

Example: Wicket injection

15.2 Scheduled Task

<scheduled-task>

<scheduled-task> schedules a job to be executed at specific times or after specific delays. The times can be specified by a cron syntax or by a simple delay parameter. The job can be either a `Runnable` bean, a method specified by an EL expression, or a URL.

When specified as an IoC bean, the bean task has full IoC capabilities, including injection, @TransactionAttribute aspects, interception and @Observes.

ATTRIBUTE	DESCRIPTION
class	the classname of the singleton bean to create
cron	a cron-style scheduling description
delay	a simple delay-based execution
init	IoC initialization for the bean
mbean-name	optional MBean name for JMX registration
method	EL expression for a method to be invoked as the task
name	optional IoC name for registering the task
period	how often the task should be invoked in simple mode

task	alternate task assignment for predefined beans

<scheduled-task> Attributes

```
element scheduled-task {
  class?
  & cron?
  & delay?
  & init?
  & mbean-name?
  & method?
  & name?
  & period?
  & task?
}
```

bean-style job configuration

The most common and flexible job configuration uses standard IoC bean-style configuration. The bean must implement `Runnable` . Like the <bean> tag, the `class` attribute specifies the `Runnable` class, and any `init` section configures the bean using Resin IoC configuration.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <scheduled-task class="qa.MyTask">
    <cron>*/5</cron>
  </scheduled-task>

</web-app>
```

Example: 5min cron bean task

task reference job configuration

The task bean can also be passed to the <scheduled-task> using a Resin-IoC EL reference. The name of the task bean would be defined previously, either in a <bean> or <component> or picked up by classpath scanning. Like the bean-style job configuration, the reference bean must implement `Runnable` .

```
<web-app xmlns="http://caucho.com/ns/resin">

  <scheduled-task task="#{taskBean}">
    <cron>0 0 *</cron>
  </scheduled-task>

</web-app>
```

Example: midnight cron bean task

method reference job configuration

<scheduled-task> can execute a method on a defined bean as the scheduler's task. The method is specified using EL reference syntax. At each trigger time, <scheduled-task> will invoke the EL method expression.

In the following example, the task invokes `myMethod()` on the `myBean` singleton every 1 hour.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <bean name="myBean" class="qa.MyBean"/>

  <scheduled-task method="#{myBean.myMethod}">
    <delay>10m</delay>
    <period>1h</period>
  </scheduled-task>

</web-app>
```

Example: 1h period method task

url job configuration

In a <web-app>, the <scheduled-task> can invoke a servlet URL at the trigger times. The task uses the servlet `RequestDispatcher` and forwards to the specified URL. The URL is relative to the <web-app> which contains the <scheduled-task.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <scheduled-task url="/cron.php">
    <cron>0 15 * * 0</cron>
  </scheduled-task>

</web-app>
```

Example: sunday cron url task

cron trigger syntax

Some ascii art from the wikipedia cron entry

```
# +---------------- minute (0 - 59)
# |  +------------- hour (0 - 23)
# |  |  +---------- day of month (1 - 31)
# |  |  |  +------- month (1 - 12)
# |  |  |  |  +---- day of week (0 - 6) (Sunday=0 or 7)
# |  |  |  |  |
  *  *  *  *  *
```

PATTERN	DESCRIPTION
*	matches all time periods
15	matches the specific time, e.g. 15 for minutes
15,45	matches a list of times, e.g. every :15 and :45
*/5	matches every `n` times, e.g. every 5 minutes
1-5	matches a range of times, e.g. mon, tue, wed, thu, fri (1-5)

cron patterns

Each field specifies a range of times to be executed. The patterns allowed are:

RANGE	EXPLANATION (USING MINUTES AS EXAMPLE)
*	run every minute
*/5	run every 5 minutes
0,5,50	run at :00, :05, :50 every hour
0-4	run at :00, :01, :02, :03, :04
0-30/2	run every 2 minutes for the first half hour

example ranges

The minutes field is always required, and the hours, days, and months fields are optional.

RANGE	EXPLANATION
0 */3	run every 3 hours
15 2 *	run every day at 0215 local time
0 0 */3	run every third day at midnight
15 0 * * 6	run every Saturday at 0015

example times

com.caucho.resources.CronResource

Often, applications need to run a task at specific times. The `CronResource` provides a standard way of doing that. Applications need only create a standard `java.lang.Runnable` task and configure the CronResource. Resin configure's the work task with bean-style configuration.

```
<web-app xmlns="http://caucho.com/ns/resin">
  <resource type="com.caucho.resources.CronResource">
    <init>
      <cron>*/15</cron>
      <work resin:type="example.PeriodicWork">
        <foo>Custom Config</foo>
      </work>
    </init>
  </resource>
</web-app>
```

Task running every 15 minutes

cron	Specifies the times the task should be run	required
work	Specifies application's work bean	required

The cron specification follows the Unix crontab format. The cron is composed of 5 fields: minutes, hours, day of month, month, and day of week.

Each field specifies a range of times to be executed. The patterns allowed are:

RANGE	**EXPLANATION (USING MINUTES AS EXAMPLE)**
*	run every minute
*/5	run every 5 minutes
0,5,50	run at :00, :05, :50 every hour
0-4	run at :00, :01, :02, :03, :04
0-30/2	run every 2 minutes for the first half hour

example ranges

The minutes field is always required, and the hours, days, and months fields are optional.

RANGE	**EXPLANATION**
0 */3	run every 3 hours
15 2 *	run every day at 0215 local time
0 0 */3	run every third day at midnight
15 0 * * 6	run every Saturday at 0015

example times

Hello, World example

```
<web-app xmlns="http://caucho.com/ns/resin">

  <scheduled-task class="example.PeriodicWork">
    <!-- every 5 minutes -->
    <cron>*/5</cron>
  </resource>

</web-app>
```

Example: WEB-INF/web.xml

```
package example;

import java.util.logging.Level;
import java.util.logging.Logger;

public class PeriodicWork implements Runnable {
  static protected final Logger log =
    Logger.getLogger(PeriodicWork.class.getName());

  public PeriodicWork()
  {
    log.info("PeriodicWork: constructor");
  }

  /**
   * Required implementation of java.lang.Runnable.run()
   */
  public void run()
  {
    log.info("PeriodicWork: run() Hello, World");
  }
}
```

Example: WEB-INF/classes/example/PeriodicWork.java

```
[13:04:27.429] PeriodicWork: constructor
[13:05:00.095] PeriodicWork: run() Hello, World
[13:10:00.182] PeriodicWork: run() Hello, World
```

Example: bean-style configuration example

```
<web-app xmlns="http://caucho.com/ns/resin">

  <scheduled-task class="example.PeriodicWork">
    <!-- every minute -->
    <cron>*</cron>

    <init>
      <message>Goodybye, World</message>
    </init>
  </scheduled-task>

</web-app>
```

WEB-INF/web.xml

```
package example;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.resource.spi.work.Work;

public class PeriodicWork implements Work {
  static protected final Logger log =
    Logger.getLogger(PeriodicWork.class.getName());

  String _message;

  public PeriodicWork()
  {
    log.info("PeriodicWork: constructor");
  }

  /**
   * Optional, called in response to presence of <message>
   * configuration tag.
   */
  public void setMessage(String message)
  {
    log.info("PeriodicWork: setMessage");
    _message = message;
  }

  /**
   * Optional, called after bean is created and any setters
   * from configuration are called.
   */
  @PostConstruct
  public void init()
    throws Exception
  {
    log.info("PeriodicWork: init()");

    if (_message == null)
      throw new Exception("`message' is required");
  }

  /**
   * Required implementation of java.lang.Runnable.run()
   */
  public void run()
  {
    log.info("PeriodicWork: run() " + _message);
  }

  /**
   * Implementation of javax.resource.spi.work.Work.release()
   */
  public void release()
  {
    log.info("PeriodicWork: release()");
  }
}
```


Example: WEB-INF/classes/example/PeriodicWork.java

```
[13:04:27.429] PeriodicWork: constructor
[13:04:27.429] PeriodicWork: setMessage
[13:04:27.429] PeriodicWork: init()
[13:05:00.095] PeriodicWork: run() Goodbye, World
[13:06:00.182] PeriodicWork: run() Goodbye, World
(close Resin)
[13:06:00.345] PeriodicWork: release()
```

RMI Resource

The goal of RMI is to provide `services` to remote clients. A remote client obtains and uses a `proxy object` that implements an `interface` . The interface is the contract for the service, it is the definition of the methods that the service provides.

Because the client is using a proxy object, the actual execution of code occurs on the server. A proxy object is placeholder that the client uses to cause execution of code on a server.

Registry

The RMI registry is used to store a list of available services. A client uses the registry to make it's proxy object, and the Registry is responsible for giving appropriate information to the client so that it can hook up with the server that implements the service.

In many scenarios, the Registry and the server for the services are in the same JVM. It is possible, however, for the Registry to run in a different JVM or even on a different machine than the server or servers that implement the services.

A registry has a TCP port that it uses to listen to incoming requests, typically this is port 1099. The RMI registry is a global resource, each JVM can have only one Registry on a particular port. This has important ramifications for the naming of services.

The Hessian alternative

If you are considering RMI as a mechanism for publishing services, you may want to consider using Hessian instead. Hessian offers the following advantages:

- it does not have a global namespace, separate web-app's can provide services with the same name without conflict
- it supports the use of clients written in languages other than Java
- it does not require the manual generation of stubs
- it does not require a security-manager

More information is available in the Hessian section of the documentation.

Requirement: security-manager

The JDK requires that a security manager be in place for the use of RMI. This is true for both clients and servers. A security manager is enabled in Resin using the configuration:

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">

  <security-manager/>

  ...
```

enabling security-manager in resin.xml

For clients that are applets, the developer does not need to enable the security manager; the browser provides the security manager.

More information is available in the Security section of the documentation.

com.caucho.resources.rmi.RmiRegistry

Resin provides the resource class to define an RMI Registry and register services with it. If the Registry is on the 'localhost' server, then Resin will also start the RMI Registry if needed.

```
<web-app>
  <resource type="com.caucho.resources.rmi.RmiRegistry">
    <init>
      <server>localhost</server>
      <port>1099</port>

      <rmi-service service-name="HelloWorld"
                   service-class="example.HelloWorldImpl"/>
      <rmi-service .... >
    </init>
  </resource>
</web-app>
```

Example use of RMI Registry

com.caucho.resources.rmi.RmiRegistry

server	the ip address of the server with the Registry	localhost
port	the port of the Registry	1099
rmi-service	an rmi service (see below)	

`RmiRegistry` is used to define the location of an RMI Registry to use. If `server` is 'localhost', then the Registry will be started on the specified port, if it has not already been started.

If `server` is something other than 'localhost', then it is assumed that the Registry has been started by some other JVM , and is treated as remote Registry to register any services defined with `rmi-server` .

rmi-service **child of:** com.caucho.resources.rmi.RmiRegistry

Each `RmiRegistry` can have `rmi-service` children, which causes the service to be instantiated and registered with the RMI Registry defined by the containing `RmiRegistry` .

service-name	the name of the service, used for registration in the Registry and also used by clients to locate the service.	required
server-class	the name of the implementation class for the service	required

Implementing a service

Interface and Implementaion An RMI service requires the developer to create two classes - an interface and an implementation. The interface defines the contract for the service, it is given to the client and it is the client view of the service. The implementation class implements the functionality; it implements the interface and is used on on the server.

The following is a simple hello world example.

```
package example;

import java.rmi.Remote;
import java.rmi.RemoteException;


public interface HelloWorld extends Remote
{
  public String sayHello()
    throws RemoteException;
}
```

interface - WEB-INF/classes/example/HelloWorld.java

```
package example;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;


public class HelloWorldImpl extends UnicastRemoteObject implements HelloWorld
{
  public HelloWorldImpl()
    throws RemoteException
  {
    super();
  }

  public String sayHello()
    throws RemoteException
  {
    return "Hello, World";
  }
}
```

implementation - WEB-INF/classes/example/HelloWorldImpl.java

Making Stubs When the client uses a service, it uses a `proxy object` . The proxy object is a placeholder, it implements the interface defined for the service, and call's through to the server so that the code is executed on the server.

RMI calls proxy objects `Stubs` , and the stubs must be manually generated. The generation of stubs is done using the `rmic` tool.

```
rmic -v1.2 -d WEB-INF/classes/  example.HelloWorldImpl
```

using rmic to generate Stubs

This call to `rmic` will use the file `WEB-INF/classes/example/HelloWorldImpl.class` to generate the class file `WEB-INF/classes/example/HelloWorldImpl_Stub.class` .

It is tedious to perform this step manually, an ant build script (as shown in a later section) can be used to expediate the process.

Deploying the service with Resin Once the work of making an interface, an implementation, and generating a stub is complete, it is a simple process to deploy the service in Resin.

```
<web-app>
  <resource type="com.caucho.resources.rmi.RmiRegistry">
    <init>
      <server>localhost</server>
      <port>1099</port>

      <rmi-service service-name="HelloWorld" service-class="example.HelloWorldImpl"/>
    </init>
  </resource>
</web-app>
```

Deploying the service with Resin

More than once service is easily deployed with the use of multiple `rmi-service` tags:

```
<web-app>
  <resource type="com.caucho.resources.rmi.RmiRegistry">
    <init>
      <server>localhost</server>
      <port>1099</port>

      <rmi-service service-name="HelloWorld" service-class="example.HelloWorldImpl"/>
      <rmi-service service-name="HelloAgainWorld" service-class="example.HelloAgainWorldImpl"/>
    </init>
  </resource>
</web-app>
```

Deploying more than once service with Resin

Choosing a name

By convention, the name chosen for the service often matches the name of the interface class. For example, if the interface name is "example.HelloWorld" then service-name is "HelloWorld" or even "example.HelloWorld" to match.

The RMI Registry has a global namespace. If two different web-app's try to publish the same service, with the same name, there will be conflicts.

An example build file An ant build file is useful for completing the `rmic` step, and for preparing a jar for use by the client. The client jar contains the interfaces and the stubs.

The following build file, placed in `/WEB-INF/build` , creates the jar file `/rmiclient.jar` .

```
<project name="rmiexample" default="dist" basedir=".">

<property file="local.properties"/>
<property file="build.properties"/>
<property environment="env"/>

<property name="build.compiler.emacs" value="true"/>
<property name="resin.home" value="${'${'}env.RESIN_HOME}"/>

<property name="rmiclient.jar" value="../rmiclient.jar"/>

<!-- NOTE: new RMI interfaces must have corresponding entries addeed
   -       in the rmiclient.jar taget
   -->

<path id="compile.classpath">
  <fileset dir="${'${'}resin.home}/lib">
    <include name="**/*.jar" />
  </fileset>
</path>

<target name="init">
  <tstamp/>
</target>

<target name="compile" depends="init">
  <mkdir dir="classes"/>
  <javac classpathref="compile.classpath"
         destdir="classes"
         debug="true">
    <src path="classes"/>
  </javac>
</target>

<target name="rmic" depends="init,compile">
  <rmic base="classes"
        classpathref="compile.classpath"
        includes="**/*Impl.class"/>
</target>


<target name="rmiclient.jar" depends="init,rmic">
  <jar destfile="${'${'}rmiclient.jar}">
    <fileset dir="classes">
      <patternset>
        <include name="**/HelloWorld.class"/>
        <include name="**/*_Stub.class"/>
      </patternset>
    </fileset>
  </jar>
</target>

<target name="dist" depends="rmiclient.jar"/>

</project>
```

Implementing a client

The client is usually on a different machine, or at least in a different JVM, than the server. That is the point of RMI, it enables the execution of code on a remote machine.

In order to use the RMI service, the client needs the interface classes and the Stubs. The easiest way to provide these to the client is to provide a jar; the ant build file above provides an example of using ant to automate the creation of the jar file for the client.

Once the jar file is available to the client, using the RMI service id fairly simple.

```
String server = "//server-with-registry.com:1099/";
HelloWorld remote = (HelloWorld) Naming.lookup(server + "HelloWorld");

System.out.println(remote.sayHello());
```

An RMI client

Scenarios

A Resin server that provides the Registry and the service In the most common scenario, the Resin server provides both the RMI Registry and the RMI services. When the registry server is defined as 'localhost', Resin will start the rmi registry if has not been started already.

This provides a simple method of using RMI, you don't have to worry about the (somewhat tricky) process of starting the rmi registry yourself.

```
<web-app>
  <resource type="com.caucho.resources.rmi.RmiRegistry">
    <init>
      <server>localhost</server>
      <port>1099</port>

      <rmi-service service-name="HelloWorld" service-class="example.HelloWorldImpl"/>
      <rmi-service .... >
    </init>
  </resource>
</web-app>
```

Scenario: a Resin server that provides the Registry and the service

When the Resin server starts, it will start the rmi registry on port 1099 and register the 'HelloWorld' service with it.

A Registry on a different server In this scenario, the rmi registry is located on the machine `services.hogwarts.com` . The registry is started with a custom (not Resin) server implemented by Hogwarts.

The requirement is for the HelloWorld service, implemented within a Resin server, to be registered with the remote Registry.

In this scenario, the Resin resource RmiRegistry is used to attach to the existing RMI registry running on `services.hogwarts.com` .

```
<web-app>
  <resource type="com.caucho.resources.rmi.RmiRegistry">
    <init>
      <server>services.hogwarts.com</server>
      <port>1099</port>

      <rmi-service service-name="HelloWorld" service-class="example.HelloWorldImpl"/>
      <rmi-service .... >
    </init>
  </resource>
</web-app>
```

A Registry on a different server

When the Resin server starts, it will register the 'HelloWorld' service with the RMI Registry on `services.hogwarts.com` . Since the server is on a remote machine, Resin will not create a registry on the local machine. When the REsin server shuts down, or is restarted, the 'HelloWorld' service will be removed from the remote registry.

A Registry in a different JVM In this scenario, the rmi registry is located on the same machine as the Resin server, but is started with a custom (not Resin) server implemented by Hogwarts.

This is essentially the same scenario as having a Registry on a different szerver. The server name cannot be provided as 'localhost', however, because Resin will try to create the RMI registry.

The solution is to use an IP address of '127.0.0.1' as the address of the server. Because the server name is not 'localhost', the RMI registry will not be created.

```
<web-app>
  <resource type="com.caucho.resources.rmi.RmiRegistry">
    <init>
      <server>127.0.0.1</server>
      <port>1099</port>

      <rmi-service service-name="HelloWorld" service-class="example.HelloWorldImpl"/>
      <rmi-service .... >
    </init>
  </resource>
</web-app>
```

A Registry in a different JVM

When the Resin server starts, it will register the 'HelloWorld' service with the RMI Registry on the local machine. Since the server is not 'localhost', Resin

will not create a registry on the local machine. When the Resin server shuts down, or is restarted, the 'HelloWorld' service will be removed from the remote registry.

Chapter 16

Amber

16.1 Amber

See Also

- See Basic tutorial for a complete single-table example.
- See Many-to-one tutorial for basic relations.

Quick Start

1. Expected SQL for the database
2. Entity bean implementation
3. Servlet loading, querying, and persisting
4. persistence.xml configuration
5. resin-web.xml configuration

```
create table HOUSE (
  id integer auto_increment,
  name varchar(255)
)
```

Example: House SQL

```
package demo;

import javax.persistence.*;

@Entity
public class House {
  @Id
  @Column(name="id")
  @GeneratedValue
  private int _id;

  @Basic
  @Column(name="name")
  private String _name;
}
```

Example: House entity

```
package demo;

import javax.ejb.*;
import javax.servlet.*;
import javax.persistence.*;

public class HouseServlet extends GenericServlet {
  @PersistenceUnit("test") EntityManagerFactory _factory;

  public void load(PrintWriter out)
  {
    EntityManager amber = _factory.createEntityManager();

    try {
      House house = amber.find(House.class, 1);

      out.println("House: " + house);
    } finally {
      amber.close();
    }
  }

  public void query(PrintWriter out)
  {
    EntityManager amber = _factory.createEntityManager();

    try {
      Query query = amber.createQuery("select o from House o WHERE o.id=1");

      out.println("House: " + query.getSingleResult());
    } finally {
      amber.close();
    }
  }

  @TransactionAttribute
  protected void insert(PrintWriter out)
  {
    EntityManager amber = _factory.createEntityManager();

    try {
      House house = new House("Gryffindor");

      amber.persist(house);
    } finally {
      amber.close();
    }
  }
}
```

Example: HouseServlet

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">

  <persistence-unit name="test">
  </persistence-unit>

</persistence>
```

META-INF/persistence.xml

```
<web-app xmlns="http://caucho.com/ns/resin">

  <ejb-server database="jdbc/test"/>

  <servlet-mapping url-pattern="/test"
                   servlet-class="demo.HouseServlet"/>

</web-app>
```

WEB-INF/resin-web.xml

API

EntityManager

```
public interface EntityManager {
  public <T> T find(Class<T> entityCLass, Object primaryKey);
  public <T> T getReference(Class<T> entityClass, Object primaryKey);

  public void flush();
  public <T> T merge(T entity);
  public void persist(Object entity);
  public void refresh(Object entity);
  public void remove(Object entity);

  public FlushModeType getFlushMode();
  public void setFlushMode(FlushModeType flushMode);

  public Query createQuery(String ql);
  public Query createNamedQuery(String name);
  public Query createNativeQuery(String sql);
  public Query createNativeQuery(String sql, Class resultClass);
  public Query createNativeQuery(String sql, String resultSEtMapping);

  public void clear();
  public void close();
  public boolean contains(Object entity);
  public Object getDelegate();
  public boolean isOpen();

  public EntityTransaction getTransaction();
  public void joinTransaction();
  public void lock(Object entity, LockModeType lockMode);
}
```

EntityManagerFactory

```
public interface EntityManagerFactory {
  public EntityManager createEntityManager();
  public EntityManager createEntityManager(Map map);

  public void close();
  public boolean isOpen();
}
```

EntityTransaction

```
public interface EntityTransaction {
  public void begin();
  public void commit();
  public void rollback();

  public boolean getRollbackOnly();
  public void setRollbackOnly();
  public boolean isActive();
}
```

Query

```
public interface Query {
  public List getResultList();
  public Object getSingleResult();
  public int executeUpdate();

  public Query setFirstResult(int startPosition);
  public Query setFlushMode(FlushModeType flushMode);
  public Query setHint(String hintName, Object value);
  public Query setMaxResults(int maxResult);

  public Query setParameter(String name, Object value);
  public Query setParameter(String name, Date date, TemporalType type);
  public Query setParameter(String name, Calendar date, TemporalType type);
  public Query setParameter(int pos, Object value);
  public Query setParameter(int pos, Date date, TemporalType type);
  public Query setParameter(int pos, Calendar date, TemporalType type);
}
```

Annotations

Class Annotations

@DiscriminatorColumn Configures the discriminator column, which select the entity class in an inheritance relationship. Each entity class will have a column value which uniquely selects the class to be loaded.

PROPERTY	DESCRIPTION	DEFAULT
name	The name of the column	
discriminatorType	The column type: STRING, CHAR or INTEGER	STRING
columnDefinition	SQL definition used when creating the column	

length	default VARCHAR length when creating a STRING column	31

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface DiscriminatorColumn {
  String name() default "";
  DiscriminatorType discriminatorType() default STRING;
  String columnDefinition() default "";
  int length() default 31;
}
```

@Embeddable Annotates the class as an embeddable value. The class fields will represent a collection of table columns embedded as part of a containing class for the table.

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Embeddable {
}
```

@Entity Annotates the class as an entity bean.

See the basic property tutorial and the basic field tutorial for an introduction.

PROPERTY	**DESCRIPTION**	**DEFAULT**
name	The name of the bean	The class name (unqualified)

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Entity {
  String name() default "";
}
```

The fields or properties will be annotated by @Id, @Basic, etc. Amber will detect either field or property annotation by the type for the @Id. In other words, if Amber sees an @Id on a field, it will use field access. If Amber sees @Id on a method, it will use property access.

@IdClass The `@IdClass` annotation specifies the class to be used to contain a compount primary key.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface IdClass {
  Class value();
}
```

@Inheritance @Inheritance marks the entity bean as supporting inheritance, i.e. the database maps to different Java classes depending on a discriminator value.

PROPERTY	DESCRIPTION	DEFAULT
strategy	The mapping strategy for inheritance: SINGLE_TABLE, JOINED or TABLE_PER_CLASS	SINGLE_TABLE

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface Inheritance {
  InteritanceType strategy() default SINGLE_TABLE;
}
```

@MappedSuperclass The `@MappedSuperclass` annotation marks the class as a parent class to an `@Entity` .

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface MappedSuperclass {
}
```

@SecondaryTable Specifies a secondary database table for an entity bean. The secondary table will contain the fields with a secondaryTable in the @Column.

PROPERTY	DESCRIPTION	DEFAULT
name	The name of the table	The unqualified class name.

catalog	the table's catalog	none
schema	the table's schema	none
pkJoinColumns	join column to the primary table	joins the primary key
uniqueConstraint	unique constraints during generation	none

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface SecondaryTable {
  String name() default "";
  String catalog() default "";
  String schema() default "";
  PrimaryKeyJoinColumn []pkJoinColumns() default {};
  UniqueConstraint []uniqueConstraints() default {};
}
```

@SequenceGenerator Specifies a sequence table to be used for generating keys.

PROPERTY	DESCRIPTION	DEFAULT
name	The amber name of the sequence table	required
sequenceName	The SQL name of the sequence table	`name`
initialValue	The initial value to seed the generator	0
allocationSize	The number of values to increment by for each allocation	50

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface SequenceGenerator {
  String name();
  String sequenceName() default "";
  int initialValue() default 0;
  int allocationSize() default 50;
}
```

@Table Specifies the database table for an entity bean. The default table name is the class name.

PROPERTY	DESCRIPTION	DEFAULT
name	The name of the table	The unqualified class name.
catalog	the table's catalog	none
schema	the table's schema	none
uniqueConstraint	unique constraints during generation	none

```
package javax.persistence;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Table {
  String name() default "";
  String catalog() default "";
  String schema() default "";
  UniqueConstraint []uniqueConstraints() default {};
}
```

@TableGenerator Specifies a secondary table to be used for generating keys.

PROPERTY	DESCRIPTION	DEFAULT
name	The amber name of the generator table	required
table	The SQL name of the generator table	`name`
catalog	The SQL catalog of the generator table	
schema	The SQL schema of the generator table	
pkColumnName	The SQL column name for the primary key name	
valueColumnName	The SQL column name for the value	
pkColumnName	The SQL column name for the primary key's value	
initialValue	The initial value to seed the generator	0
allocationSize	The number of values to increment by for each allocation	50

uniqueConstraints	Extra uniqueness constraints when creating the table

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface TableGenerator {
  String name();
  String table() default "";
  String catalog() default "";
  String schema() default "";
  String pkColumnName() default "";
  String valueColumnName() default "";
  String pkColumnValue() default "";
  int initialValue() default 0;
  int allocationSize() default 50;
  UniqueConstraint []uniqueConstraints() default {};
}
```

Property Annotations

@Basic Marks a field as a persistent field.

PROPERTY	DESCRIPTION	DEFAULT
fetch	EAGER or LAZY fetching	FetchType.EAGER
optional	if true, the column may be null	true

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Basic {
  FetchType fetch() default EAGER;
  boolean optional() default true;
}
```

The fetch types are:

- EAGER - fetch the field when the bean is loaded
- LAZY - fetch the field only when the field is used

```
@Entity
public class Course {
  @Basic
  public String getName()

  ...
}
```

Example: string property

```
@Entity
public class Course {
  @Basic(fetch=FetchType.LAZY)
  public String getMassiveText()

  ...
}
```

Example: lazy-loaded property

@Column Specifies the field's SQL column name as well as any CREATE TABLE properties for auto generation.

PROPERTY	DESCRIPTION	DEFAULT
name	The SQL name of the column	the field name
unique	True for UNIQUE columns	false
nullable	False for IS NOT NULL columns	true
insertable	True if column is inserted on as SQL `INSERT` call	true
updatable	True if column is updated when the field is modified	false
columnDefinition	SQL to create the column in a CREATE TABLE	none
table	specified if column is stored in a secondary table	none
length	the default length for a VARCHAR for a CREATE TABLE	255

precision	the default length for a number definition for a CREATE TABLE	0
scale	the default length for a number definition for a CREATE TABLE	0

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Column {
  String name() default "";
  boolean unique() default false;
  boolean nullable() default true;
  boolean insertable() default true;
  boolean updateable() default true;
  String columnDefinition() default "";
  String table() default "";
  int length() default 255;
  int precision() default 0;
  int scale() default 0;
}
```

```
@Entity
public class Course {
  @Basic
  @Column(name="MY_NAME",
          unique=true,
          nullable=false,
          length=32)
  public String getName()

  ...
}
```

Example: @Column for a string property

@Embedded Marks a field as containing an embeddable value. The field's value will be a class marked as @Embeddable. Applications can use @Embedded fields to gather columns into meaningful groups.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Embedded {
}
```

@EmbeddedId Marks a field as a primary key with a embedded class. The field's value class must be marked with `@Embeddable` . Applications can use `@EmbeddedId` to implement compound keys.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface EmbeddedId {
}
```

@Enumerated Marks a field as containing an enumerated value.

PROPERTY	DESCRIPTION	DEFAULT
value	Specifies whether the enum's ORDINAL or STRING representation should be saved in the database.	ORDINAL

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Enumerated {
  EnumType value() default ORDINAL;
}
```

@GeneratedValue Used with @Id to specify a generator for automatic key generation when new objects are created.

PROPERTY	DESCRIPTION	DEFAULT
generator	The sequence or table generator name	${table}_cseq
strategy	The auto-generation type: TABLE, SEQUENCE, IDENTITY or AUTO	AUTO

The generator types are:

- IDENTITY - the database supplies the new key, e.g. auto_increment, SERIAL, or IDENTITY

- SEQUENCE - use a SEQUENCE type to generate the key
- TABLE - use a @TableGenerator for the key
- AUTO - choose the generator based on the database
  - MySQL - IDENTITY using auto_increment
  - Resin - IDENTITY using auto_increment
  - Postgres - SEQUENCE
  - Oracle - SEQUENCE

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface GeneratedValue {
  GenerationType strategy() default AUTO;
  String generator() default "";
}
```

For SEQUENCE and TABLE, Resin will create the sequence name as "${table}_cseq".

```
import javax.persistence.*;

@Entity
public class Course {
  @Id
  @GeneratedValue
  public long getId()

  ...
}
```

Example: autoincrement generation

```
import javax.persistence.*;

@Entity
public class Course {
  @Id
  @GeneratedValue(strategy=GeneratorType.AUTO
                  generator="COURSE_SEQ")
  public long getId()

  ...
}
```

Example: sequence generation

@Id Marks a field as a primary key. The `@Id` may be used in combination with `@GeneratedValue` to specify a generator for automatic key generation when new objects are created.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Id {
}
```

The default column name is "ID".

```
import javax.persistence.*;

@Entity
public class Course {
  @Id
  @Column(name="t_id")
  @GeneratedValue
  public long getId()

  ...
}
```

Example: automatic generation

@Lob Marks a field as containing a large blob value.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Lob {
}
```

@Temporal Marks a field as a time-based value.

PROPERTY	DESCRIPTION	DEFAULT
value	The SQL type used for the field value: DATE, TIME or TIMESTAMP	TIMESTAMP

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Temporal {
  TemporalType value() default TIMESTAMP;
}
```

@Transient `@Transient` makes a field non-persistent. `@Transient` fields work like the `transient` annotation to prevent properties being saved.

```
@Target({METHOD,FIELD})
@Retention(RUNTIME)
public @interface Transient {
}
```

@Version `@Version` marks a version field, used for optimistic locking.

```
@Target({METHOD,FIELD})
@Retention(RUNTIME)
public @interface Version {
}
```

Relation annotations

@JoinColumn Defines a join (foreign) columns. Used for @ManyToOne.
See also @Column for corresponding definition for @Basic columns.
See the Many-to-One tutorial for a full example.

PROPERTY	DESCRIPTION	DEFAULT
name	The column name of the source table	the column name of the target key
referencedColumnName	The target column for composite keys	the single primary key
unique	True if unique	false
nullable	False if IS NOT NULL	true
insertable	True if the column is inserted on a `create`	true
updateable	True if the column is updated on field changes	true
columnDefinition	SQL column definition	false
table	specifies a secondary table if not in the primary	none

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface JoinColumn {
  String name() default "";
  String referencedColumnName() default "";
  boolean unique() default false;
  boolean nullable() default false;
  boolean insertable() default true;
  boolean updateable() default true;
  String columnDefinition() default "";
  String table() default "";
}
```

```
public class Student {
  @Id
  @Column(name="student_id")
  long getId()

  @ManyToOne
  @JoinColumn(name="house_id")
  public House getHouse()
}
```

Example: Student to House link

```
CREATE TABLE Student {
  student_id BIGINT PRIMARY KEY auto_increment

  house_id BIGINT REFERENCES House(id)
)
```

Example: Student SQL

@JoinColumns Defines a set of join (foreign) columns for composite keys.

```
@Target({TYPE,METHOD, FIELD})
@Retention(RUNTIME)
public @interface JoinColumns {
  JoinColumn [] value() default{}
}
```

@JoinTable Defines an association table for a many-to-many relation.

PROPERTY	DESCRIPTION	DEFAULT
name	Table definition for the association table	concatening the source and target table names
catalog	Database catalog	""

schema	Database schema	""
joinColumns	Columns from from the association table to the source table	Uses the source table primary key
inverseJoinColumns	Columns from from the association table to the target table	Uses the target table primary key

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface JoinTable {
  String table() default "";
  String catalog() default "";
  String schema() default "";
  JoinColumn []joinColumns() default {};
  JoinColumn []inverseJoinColumns() default {};
  UniqueContraint []uniqueConstraint() default {};
}
```

@ManyToMany Marks a field as a many-to-many (association) relation. The column names are the key columns of the source and target tables. See the many-to-many tutorial for an example.

PROPERTY	DESCRIPTION	DEFAULT
cascade	Operations which cascade to the target	none
fetch	EAGER or LAZY fetching	FetchType.EAGER
mappedBy	Specifies the source relation if a target	
targetEntity	The class of the target entity	the property's type

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ManyToMany {
  String targetEntity default "";
  CascadeType []cascade() default {};
  FetchType fetch() default LAZY;
  String mappedBy isInverse() default "";
}
```

```
@Entity
public class Student {
  @ManyToMany
  @JoinTable(
    name="student_course_map",
    joinColumns={@JoinColumn(name="student_id")},
    inverseJoinColumns={@JoinColumn(name="course_id")}
  )
  public Collection getCourses()

  ...
}
```

Example: @ManyToMany link

@ManyToOne Marks a field as a many-to-one (link) relation.
The default column name is the column name of the target key.
See the many-to-one tutorial for an example.

PROPERTY	DESCRIPTION	DEFAULT
targetEntity	The class of the target entity	the property's type
cascade	Operations which cascade to the target: ALL, PERSIST, MERGE, REMOVE or REFRESH	none
fetch	EAGER or LAZY fetching	FetchType.EAGER
optional	If false, the relation must always have a value	true

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ManyToOne {
  String targetEntity default "";
  CascadeType []cascade() default {};
  FetchType fetch() default EAGER;
  boolean optional() default true;
}
```

```
@Entity
public class Student {
  @ManyToOne
  @JoinColumn(name="house")
  public House getHouse()

  ...
}
```

Example: @ManyToOne link

@MapKey Marks a field as key in a Map relationship.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface MapKey {
  String name() default "";
}
```

@OneToMany Marks a field as a one-to-many (collection) relation. Because a one-to-many field is dependent, it needs a @ManyToOne relation on the source table which defines the column.

PROPERTY	DESCRIPTION	DEFAULT
targetEntity	The class of the target entity	the property's type
cascade	Operations which cascade to the target: ALL, PERSIST, MERGE, REMOVE, and REFRESH	none
fetch	EAGER or LAZY fetching	FetchType.EAGER
mappedBy	Specifies the owning @ManyToOne property on the target entity.	

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OneToMany {
  String targetEntity default "";
  CascadeType []cascade() default {};
  FetchType fetch() default EAGER;
  String mappedBy() default "";
}
```

```
@Entity
public class House {
  ...
  @OneToMany(targetEntity=Student.class,
             mappedBy="house")
  public Collection getStudents()
}

@Entity
public class Student {
  ...
  @ManyToOne
  @JoinColumn(name="house")
  public House getHouse()
}
```

Example: collection

```
CREATE TABLE House {
  id BIGINT PRIMARY KEY
)

CREATE TABLE Student {
  id BIGINT PRIMARY KEY,

  house BIGINT REFERENCES House(id)
)
```

Example: Collection SQL

@OneToOne Marks a field as a one-to-one (dependent link) relation. Because a one-to-one field is dependent, it needs a @ManyToOne relation on the source table which defines the column.

PROPERTY	DESCRIPTION	DEFAULT
targetEntity	The class of the target entity	the property's type

cascade	Operations which cascade to the target: ALL, PERSIST, MERGE, REMOVE, and REFRESH	none
fetch	EAGER or LAZY fetching	FetchType.EAGER
mappedBy	Specifies the owning relation	

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OneToOne {
  String targetEntity default "";
  CascadeType []cascade() default {};
  FetchType fetch() default EAGER;
  boolean optional() default true;
  String mappedBy() default "";
}
```

@OrderBy `@OrderBy` specifies the SQL column to use for ordering collection relations.

PROPERTY	DESCRIPTION	DEFAULT
name	The property name to sort the collection by.	the property's type

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OrderBy {
  String value() default "";
}
```

Amber Lifecycle

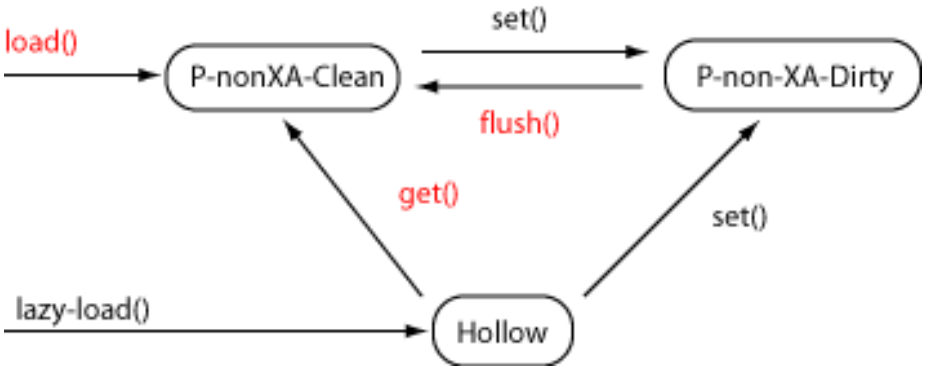
Non-Transactional Lifecycle

Amber's non-transactional lifecycle has three important states:

- **clean:** the bean is loaded from the database
- **dirty:** the bean has unwritten changes

- **hollow:** the bean is unloaded (lazily-loaded)

In the diagram below, the red methods (`load()` , `getXXX()` , and `flush()`) query and update the database.



The `aConn.load("1")` method loads the bean from the database and transitions to the `clean` state.

Calling `test.setData("foo")` will change to the `dirty` state.

Calling `aConn.flush()` writes the changes to the database and changes to the `clean` state. Amber may also flush the changes and change to the clean state at any time. `flush()` merely guarantees that the changes will be flushed to the database.

The `hollow` state represents lazily-loaded entities. many-to-one relations and some queries will return the unloaded bean instead of a loaded bean. When the application calls a `getXXX()` method, the bean will load from the database and change to the `clean` state. When the application calls a `setXXX()` method, the bean will change to the `dirty` state.

```
public class MyServlet extends GenericServlet {
  @In EntityManagerFactory _factory;
  @In UserTransaction _trans;

  ...

  public void doTest(PrintWriter out)
    throws IOException
  {
     EntityManager aConn = _factory.createManager();

     // load() loads test and then detaches it
     qa.Test test = aConn.load(qa.Test.class, "1");

     // test has the loaded values
     out.println(test.getData());

     // but parent is not lazily-loaded when detached, i.e. it's null.
     qa.Test parent = test.getParent();

     aConn.close();
  }
}
```

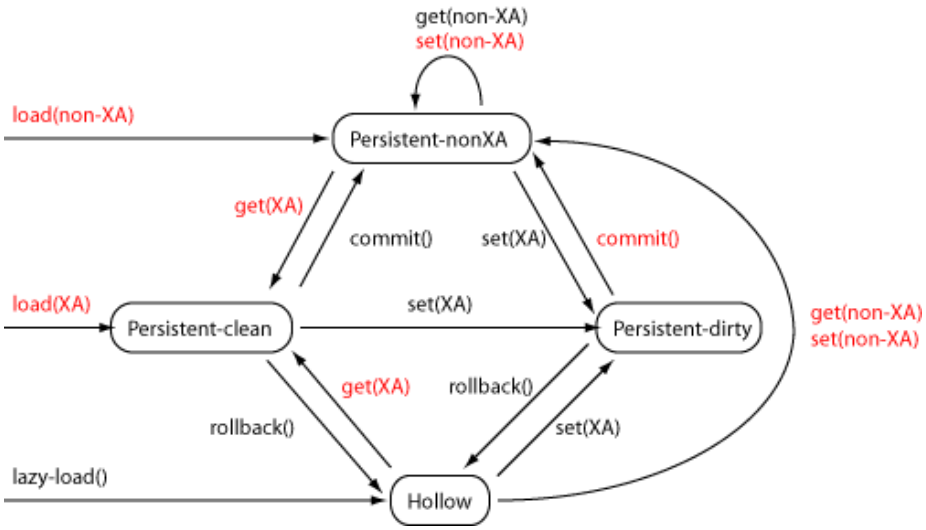
Example: Amber outside transaction

Transactional Lifecycle

In a transaction, Amber loads the bean from the database, even if it was loaded outside of the transaction. (Exceptions exist for cases like read-only beans.) By loading the bean in the transaction, Amber lets the database handle the transactional locking and state consistency.

Just like the non-transactional `clean` and `dirty` states, Amber has transactional `clean` and `dirty` states called `Persistent-clean` and `Persistent-dirty` . As in the non-transactional case, the `hollow` state represents lazily-loaded beans.

- **persistent-clean:** the bean is loaded from the database within the transaction
- **persistent-dirty:** the bean has been changed
- **hollow:** the bean is unloaded (lazily-loaded or rolled-back)
- **persistent-nonXA:** the bean was loaded outside of the transaction (and would need reloading if used in the transaction)



The main differences from the non-transactional lifecycle are:

- Transactions need a load from inside the transaction. Loads before the transaction cannot be reused.

- Updates occur during the commit() call and change to the nonXA-clean state

- Rollbacks change to the hollow state.

Configuration Files

Example configuration

The lifecycle description uses a single running example, Test, which has two properties: `getData()` which returns a string, and `getParent()` which is a pointer to another Test object.

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
   version="1.0">
  <package>qa</package>
  <entity name="Test" class="qa.Test" access="PROPERTY">
    <table name="TEST"/>
      <attributes>
        <id name="id">
          <column name="ID"/>
        </id>
        <basic name="data">
          <column name="DATA"/>
        </basic>
        <many-to-one name="parent">
          <join-column name="FK_PARENT"/>
        </many-to-one>
     </attributes>
   </table>
  </entity>
</entity-mappings>
```

Example: META-INF/orm.xml

Chapter 17

Embedding Resin

17.1 Embedding Resin

Example: creating a standalone web server

1. Download Resin from http://caucho.com/download

2. Unzip Resin in /usr/local/share and make a symlink from /usr/local/share/resin

3. Add the jars in resin/lib/*.jar to the CLASSPATH

4. Create and compile a `TestResin` class as described below

5. Browse http://localhost:8080

```
package example;

import com.caucho.resin.*;

public class TestResin {

  public static void main(String []args)
  {
    ResinEmbed resin = new ResinEmbed();

    HttpEmbed http = new HttpEmbed(8080);
    resin.addPort(http);

    WebAppEmbed webApp = new WebAppEmbed("/", "/var/www/htdocs");
    resin.addWebApp(webApp);

    resin.start();
    resin.join();
  }
}
```

Example: example/TestResin.java

Example: embedding Resin for testing

For testing, you can create an embedded Resin instance and browse URLs programmatically using the `resin.request()` method. Since the `request` method runs Resin's normal HTTP processing, you can use any HTTP requests or headers.

```
package example;

import com.caucho.resin.*;

public class TestResin {

  public void main(String []args)
  {
    ResinEmbed resin = new ResinEmbed();

    WebAppEmbed webApp = new WebAppEmbed("/", "/var/www/htdocs");
    resin.addWebApp(webApp);

    resin.start();

    String result = resin.request("GET /test.jsp");
    System.out.println(result);
  }
}
```

Example: example/TestResin.java

ResinEmbed

The ResinEmbed class represents a Resin instance. It contains:

- A set of ports (usually http)
- A set of beans available through WebBeans injection
- A set of web-apps
- Methods for starting/stopping
- Methods for Java-based HTTP requests

```
public class ResinEmbed {
  public ResinEmbed();
  public ResinEmbed(String resinConfPath);

  public void addBean(BeanEmbed bean);
  public void addPort(PortEmbed port);
  public void setServerHeader(String serverName);
  public void addWebApp(WebAppEmbed webApp);

  public void join();
  public void destroy();
  public void start();

  public void request(InputStream is, OutputStream os)
    throws IOException;
  public void request(String request, OutputStream os)
    throws IOException;
  public String request(String request)
    throws IOException;
}
```

A `ResinEmbed` can be created and started without any other classes, although it won't do anything useful. The following example will return a *404 Not Found* response string from the request since there are no web-apps configured. The example will not listen to any ports at all, since no `HttpEmbed` objects have been added.

```
public static void main(String [])
{
  ResinEmbed resin = new ResinEmbed();

  resin.start();

  String result = resin.request("GET /test.jsp");

  System.out.println(result);
}
```

Example: Trivial ResinEmbed call

See also the ResinEmbed JavaDoc.

BeanEmbed

Beans are created using the `BeanEmbed` API. If your application wants to expose services to the embedded web-app, just add a BeanEmbed to the Resin instance. `BeanEmbed` can also create dynamically created service, by setting a class name instead of an object.

```
public class BeanEmbed {
  public BeanEmbed();
  public BeanEmbed(Object value);
  public BeanEmbed(Object value, String name);
  public BeanEmbed(String className, String name);

  public void setClass(String className);
  public void setName(String name);
  public void setValue(Object value);

  public void addProperty(String name, Object value);
}
```

```
public void main()
{
  MyService service = new MyService();

  ResinEmbed resin = new ResinEmbed();

  resin.addBean(new BeanEmbed(service, "my-service"));

  resin.addWebApp(new WebAppEmbed("/", "/var/www/htdocs"));
  resin.start();

  String result = resin.request("GET /test.php");

  System.out.println(result);
}
```

Example: Adding Bean services

A testing PHP file could use `java_bean()` to retrieve the service.

```
my-service <?= java_bean("my-service") ?>
```

Example: test.php

A testing servlet can inject the service with `@javax.webbeans.In` .

```
package qa;

import javax.servlet.*;
import javax.webbeans.*;

public class MyServlet extends GenericServlet {
  @In MyService _myService;

  ...
}
```

Example: qa/MyServlet.java

The following example configures a dynamically-created bean instance and adds some <init> property values.

```
BeanEmbed bean = new BeanEmbed("example.MyBean");

bean.setName("my-bean");

bean.addProperty("greeting", "hello, world");

webApp.addBean(bean);
```

Example: Dynamic Bean

HttpEmbed

You can add http ports using the HttpEmbed class. When you start Resin, it will listen to the configured ports.

```
public class HttpEmbed {
  public HttpEmbed();
  public HttpEmbed(int port);
  public HttpEmbed(int port, String ipAddress);
}
```

The following trivial example will start Resin as the web server listening to port 8080 and always returning 404 since there are no web-apps defined.

```
public static void main(String [])
{
  ResinEmbed resin = new ResinEmbed();

  HttpEmbed http = new HttpEmbed(8080);

  resin.addHttp(http);

  resin.start();
  resin.join();
}
```

Example: Trivial HttpEmbed call

WebAppEmbed

WebAppEmbed represents a web-app. The defaults are the same as if Resin was started normally, i.e. the standard file, jsp, and php servlets are already defined, and will read the WEB-INF/web.xml and WEB-INF/resin-web.xml (and compile classes in WEB-INF/classes). Normally, an embedded web-app will just set the context-path, root-directory and possibly add extra beans, although it's possible to add servlets and filters as well.

- The context-path (i.e. the URL prefix)
- The root-directory
- An optional archive-path for a .war file
- Any added *BeanEmbed* beans
- Any added *ServletMappingEmbed* servlets
- Any added *FilterMappingEmbed* filters

For unit testing, you can use combination of *BeanEmbed* and test web-app directories as a unit test framework. Each *test-x.php* (or qa.TestServletX) can test a different aspect of the service.

```
public class WebAppEmbed {
  public WebAppEmbed();
  public WebAppEmbed(String contextPath);
  public WebAppEmbed(String contextPath, String rootDirectory);

  public void setArchivePath(String archivePath);
  public String getArchivePath();
  public String getContextPath();
  public void setContextPath(String contextPath);
  public String getRootDirectory();
  public void setContextParam(String name, String value);

  public void addBean(BeanEmbed bean);

  public void addFilter(FilterEmbed servlet);
  public void addFilterMapping(FilterMappingEmbed mapping);

  public void addServlet(ServletEmbed servlet);
  public void addServletMapping(ServletMappingEmbed mapping);
}
```

```
public void main()
{
  MyService service = new MyService();

  ResinEmbed resin = new ResinEmbed();

  WebAppEmbed webApp = new WebAppEmbed("/", "/home/qa/test1");
  webApp.addBean(new BeanEmbed(service));

  resin.addWebApp(webApp);

  resin.start();

  String result = resin.request("GET /test-a.php");
  System.out.println(result);

  result = resin.request("GET /test-b.php");
  System.out.println(result);
}
```

Example: Adding Bean to a web-app

ServletMappingEmbed

`ServletMappingEmbed` lets you configure servlets in an embedded Resin instance, e.g. if you want to expose an administration application or deployment application and not put these in the resin-web.xml. `ServletMappingEmbed` lets you configure <init-param> values as well as <init> properties.

```
public class ServletMappingEmbed {
  public ServletMappingEmbed();
  public ServletMappingEmbed(String servletName);
  public ServletMappingEmbed(String servletName, String urlPattern);
  public ServletMappingEmbed(String servletName, String urlPattern,
                             String servletClass);

  public String getServletClass();
  public void setServletClass(String servletClass);
  public String getServletName();
  public void setServletName(String servletName);
  public String getUrlPattern();
  public void setUrlPattern(String urlPattern);

  public void setLoadOnStartup(int loadOnStartup);
  public void setInitParam(String name, String value);
  public void addProperty(String name, Object value);

  public void setProtocol(ServletProtocolEmbed protocol);
}
```

```
webApp = new WebAppEmbed("/", "/home/qa/test1");

servlet = new ServletMappingEmbed("my-servlet", "/test", "example.MyServlet");
webApp.addServletMapping(servlet);
```

Adding Servlet to a web-app

ServletProtocolEmbed

`ServletProtocolEmbed` lets you export remote services using Hessian, Burlap, or any other protocol implementation which provides a driver for Resin. The configuration for a remote service is exactly the same as for a servlet, i.e. using `ServletMappingEmbed` , and just adds a `ServletProtocolEmbed` to select the protocol.

```
public class ServletProtocolEmbed {
  public ServletProtocolEmbed();
  public ServletProtocolEmbed(String uri);

  public void setUri(String uri);
  public void addProperty(String name, Object value);
}
```

```
service = new ServletMappingEmbed("my-service", "/hessian",
                                  "example.MyService");

protocol = new ServletProtocolEmbed("hessian");

service.addProtocol(protocol);

webApp.addServlet(service);
```

Example: Hessian service

jUnit

- See wiki: jUnit and Resin

```
package qa;

import org.junit.*;
import static org.junit.Assert.*;
import com.caucho.resin.*;

public class MyTest {
  private static ResinEmbed _resin;

  @BeforeClass
  public static void setup()
  {
    _resin = new ResinEmbed();
    WebAppEmbed webApp = new WebAppEmbed("/", "file:/tmp/caucho/qa/test");
    _resin.addWebApp(webApp);
    _resin.start();
  }

  @Test
  public void test1plus1()
    throws java.io.IOException
  {
    assertEquals(_resin.request("GET /test.php?a=1&b=1"), "1 + 1 = 2");
  }

  @Test
  public void test1plus2()
    throws java.io.IOException
  {
    assertEquals(_resin.request("GET /test.php?a=1&b=2"), "1 + 2 = 3");
  }

  @AfterClass
  public static void shutdown()
  {
    if (_resin != null)
      _resin.destroy();
  }
}
```

Example: jUnit test

command-line, ResinEmbed main()

The `ResinEmbed` class contains a `main()` method which can be used to launch an trivial instance of Resin. Its main use is for IDEs which want to launch a testing instance of Resin.

–port=8080	TCP port to listen to
–deploy:	Enables the local deployment webapp for IDEs

command-line arguments

```
resin> java -classpath $CP com.caucho.resin.ResinEmbed --port=8080
```

Example: launching embedded Resin from the command line

Chapter 18

Filters

18.1 Filters

Filter Configuration

Filter configuration follows the Servlet 2.3 deployment descriptors. Creating and using a filter has three steps:

1. Create a filter class which extends javax.servlet.Filter
2. Use <filter> in the web.xml to configure the filter.
3. Use <filter-mapping> to select URLs and servlets for the filter.

Some other pages which discuss filters include:

- Resin's filter library

filter

Defines a filter alias for later mapping.

TAG	MEANING
filter-name	The filter's name (alias)
filter-class	The filter's class (defaults to servlet-name)
init-param	Initialization parameters

The following example defines a filter alias 'image'

```
<web-app id='/'>

<filter-mapping url-pattern='/images/*'
                filter-name='image'/>

<filter filter-name='image'
         filter-class='test.MyImage'>
  <init-param title='Hello, World'/>
</filter>

</web-app>
```

filter-name

Alias of the filter.

filter-class

Class of the filter. The CLASSPATH for filters includes the WEB-INF/classes directory and all jars in the WEB-INF/lib directory.

init-param

Initializes filter variables. `filter-param` defines initial values for `getFilterConfig().getInitParameter("foo")` .

The full Servlet 2.3 syntax for init-param is supported and allows a simple shortcut

```
<web-app id='/'>

<filter filter-name='test.HelloWorld'>
  <init-param foo='bar'/>

  <init-param>
    <param-name>baz</param-name>
    <param-value>value</param-value>
  </init-param>
</filter>

</web-app>
```

filter-mapping

Maps url patterns to filters. `filter-mapping` has two children, `url-pattern` and `filter-name` . `url-pattern` selects the urls which should execute the filter.

`filter-name` can either specify a servlet class directly or it can specify a servlet alias defined by `filter` .

CONFIGURATION	DESCRIPTION
url-pattern	A pattern matching the url: `/foo/*` , `/foo` , or `*.foo`
url-regexp	A regular expression matching the url
servlet-name	A servlet name to match.
filter-name	The filter name
filter-class	The filter class
init-param	Initialization parameters

```
<web-app xmlns="http://caucho.com/ns/resin">

<servlet servlet-name='hello'
         servlet-class='test.HelloWorld'/>

<servlet-mapping url-pattern='/hello'
                 servlet-name='hello'/>

<filter filter-name='test-filter'
        filter-class='test.MyFilter'/>

<filter-mapping url-pattern='/hello/*'
                filter-name='test-filter'/>

<filter-mapping servlet-name='hello'
                filter-name='test.SecondFilter'/>

</web-app>
```

Example: WEB-INF/resin-web.xml

GzipFilter

The GzipFilter compresses the output of pages for browsers which understand compression, and leaves the output unchanged if the browser does not support compression. A browser indicates to the server that it supports gzip compression by including "gzip" in the "Accept-Encoding" request header.

GzipFilter is available in Resin-Professional.

use-vary	Set the standard HTTP "Vary" response header to "Accept-Encoding". This indicates to the browser and any intervening cache that the response from this url might be different depending on the Accept-Encoding provided by the browser.	true
no-cache	Forbid the browser and any intervening cache from caching the results of this request. Sets the "Cache-Control" response header to "no-cache". If use-vary is false then this is always true.	false
embed-error-in-output	Embed the stack trace of any exception into the output stream that is being sent to the browser.	false

```
<web-app xmlns="http://caucho.com/ns/resin">
  <filter filter-name="gzip"
          filter-class="com.caucho.filters.GzipFilter"/>

  <filter-mapping url-pattern="/*" filter-name="gzip"/>
</web-app>
```

See .

XsltFilter

The XsltFilter transforms the response using xslt. A Servlet or a JSP can produce XML results which are transformed using xslt.

```
<web-app xmlns="http://caucho.com/ns/resin">
  <filter filter-name="xslt"
          filter-class="com.caucho.filters.XsltFilter"/>

  <filter-mapping url-pattern="/*.jsp" filter-name="xslt"/>
</web-app>
```

unconditional	always do a transformation, regardless of the content type.	false

See .

A request attribute or xml processing directive specifies the stylesheet

Resin's XsltFilter determines the stylesheet (*.xsl file) to apply from the first of:

1. The value of `request.getAttribute("caucho.xsl.stylesheet")`
2. A stylesheet specified in the source document with <?xml-stylesheet href='...'?>
3. `default.xsl`

The classpath is used to find the stylesheet. A stylesheet can be placed in `WEB-INF/classes/` , or a specific `xsl` directory can be indicated:

```
<web-app xmlns="http://caucho.com/ns/resin">
  <class-loader>
    <simple-loader path="WEB-INF/xsl"/>
  </class-loader>

  ...
</web-app>


WEB-INF/xsl/default.xsl
```

Classpath for xsl stylesheets

In a JSP the request attribute can be set with the following:

```
<% request.setAttribute("caucho.xsl.stylesheet","transform.xsl"); %>
```

request attribute to indicate the stylesheet

Specifying a processing instruction is another way to indicate the stylesheet (but a bit slower for performance considerations):

```
<?xml-stylesheet type="text/xsl" href="transform.xsl"?>
```

processing instruction to indicate the stylesheet

The stylesheet is applied only for certain content types

The filter by default will only transform responses that have a content type of one of the following:

- x-application/xslt
- x-application/xsl
- x-application/stylescript

The filter can also be configured to unconditionally do a transformation regardless of the content type:

```
<web-app xmlns="http://caucho.com/ns/resin">
  <filter filter-name='xslt' filter-class='com.caucho.filters.XsltFilter'>
    <init>
      <unconditional>true</unconditional>
    </init>
  </filter>

  <filter-mapping url-pattern='/xslt/*' filter-name='xslt'/>
</web-app>
```

TransactionFilter

The TransactionFilter wraps the request in a UserTransaction and commits the transaction when the servlet completes. All database calls for the request will either succeed together or fail. The UserTransaction is obtained by doing a jndi lookup with the name "java:comp/UserTransaction".

This filter will gracefully handle any exceptions that occur during the request by doing a rollback on the transaction.

```
<web-app xmlns="http://caucho.com/ns/resin">
  <filter filter-name='transaction-filter'
          filter-class='com.caucho.filters.TransactionFilter'/>

  <filter-mapping url-pattern='/DatabaseServlet/*'
                  filter-name='transaction-filter'/>
</web-app>
```

See .

ExpiresFilter

The ExpiresFilter sets the Expires cache control header, allowing servlet output and jsp results to be cached for a short time.

This is useful for indicating an Expires time to the browser, and it is even more useful when used in conjunction with Resin's HTTP proxy cache. If Resin's HTTP proxy cache is in use, even a short Expires time (like 2s) can result in performance gains.

cache-time	The amount of time before the servlet output will be requested again. In seconds (2s), minutes (2m), hours (2h), or days (2D).	2s

```
<web-app xmlns="http://caucho.com/ns/resin">
  <filter filter-name='expires-60s'
          filter-class='com.caucho.filters.ExpiresFilter'>
    <init>
      <cache-time>60s</cache-time>
    </init>
  </filter>

  <filter-mapping servlet-name='StockQuoteServlet'
                  filter-name='expires-60s'/>
</web-app>
```

Caching stock quotes for 60 seconds

In this example, the StockQuoteServlet will be only called once every 60 seconds. This indicates to a browser that it should refresh it's local cache of the url after 60 seconds have passed. If Resin's HTTP proxy cache is being used, the first request from any browser will cause execution of the StockQuoteServlet,

any requests from any browser for the next 60 seconds will be served from the proxy cache (the servlet will not be called).

See .

AnonymousExpiresFilter

The AnonymousExpiresFilter caches the response for anonymous users. A user is anonymous if the do not have a session. When a page has custom formatting for logged in users, it may still want to cache the results for non-logged in users saving time and database access.

The benefits of using this filter are similar to those described in ExpiresFilter.

cache-time	The amount of time before the servlet output will be requested again. In seconds (2s), minutes (2m), hours (2h), or days (2D).	2s

Servlets should call `request.getSession(false)` to get their sessions, because once a session is created the response will no longer be cached. For the same reason, JSP pages should set `<jsp:directive.page session='false'/>` for all pages that do not need a session.

```
<web-app xmlns="http://caucho.com/ns/resin">
  <filter filter-name='anonymous-expires'
          filter-class='com.caucho.filters.AnonymousExpiresFilter'>
    <init>
      <cache-time>15m</cache-time>
    </init>
  </filter>

  <filter-mapping url-pattern='*.jsp'
                  filter-name='anonymous-expires'/>
</web-app>
```

Caching all anonymouse users *.jsp pages for 15 minutes

See .

RewriteFilter

The RewriteFilter rewrites and forwards URLs matching a regular expression. It is useful either when URLs change during a site redesign or when a site might want to hide its JSP structure. The functionality is similar to mod_rewrite in the Apache web server.

The RewriteFilter is configured with a list of <rewrite> tags. Each tag has a `pattern` which matches against the URL and a `target` which specifies the new URL to be forwarded to. Multiple <rewrite> tags are allowed.

```
<filter filter-name='rewrite'
        filter-class='com.caucho.filters.RewriteFilter'>
  <init>
    <rewrite pattern="/a/([^/]*)/([^?]*)" target="/$2/$1.jsp"/>
    <rewrite pattern="/b/([^/]*)/([^?]*)" target="/$2/$1.html"/>
  </init>
</filter>

<filter-mapping url-pattern='/*' filter-name='rewrite'/>
```

RewriteFilter example

See .

ThrottleFilter

The ThrottleFilter filter implemented with restricts the number of requests from the same IP, defaulting to 2 (the HTTP spec limit.) The ThrottleFilter is useful to limit some parallel download programs that can use more threads than they should.

```
<filter filter-name="throttle"
        filter-class="com.caucho.filters.ThrottleFilter">
  <init>
    <max-concurrent-requests>2</max-concurrent-requests>
  </init>
</filter>

<filter-mapping url-pattern="/*" filter-name="throttle"/>
```

Chapter 19

BAM

19.1 BAM

Overview

- **Jabber compatible:** Since BAM is a generalization of the Jabber IM architecture, Jabber services like IM, multi-user chat, pub/sub are straightforward to implement with the BAM api.
- **Streaming:** At the core, BAM sends unidirectional messages through a single `BamStream` interface for clients, services, and the broker. The streaming architecture allows for filters, queuing, translators, and routing for sophisticated applications.
- **Federated addressing:** BAM requests are addressable like email or Jabber IM messages, so applications log on once to their local broker, and send messages to any service in the BAM network.
- **Model-based payloads:** Messages in BAM can be any `Serializable` object, letting developers choose appropriate object models for their messages.
- **Typed, mixin services:** Since messages are typed, services are designed around a mixin architecture. Each set of messages provides a sub-service. So a single service address might provide chat, pub/sub and custom query capabilities just by providing handlers for each message type.

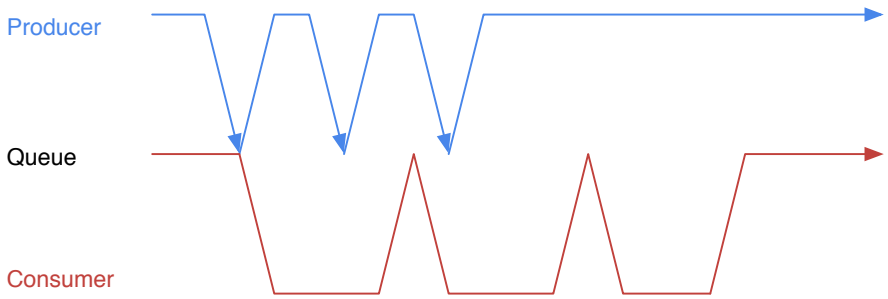
Quick Start Examples

Queued Messages

Sending a message from a client to a named service is a simple and important use of BAM. Typical applications include message queuing, chat text, game updates, Atom/RSS updates, pub/sub messaging, and event notification for

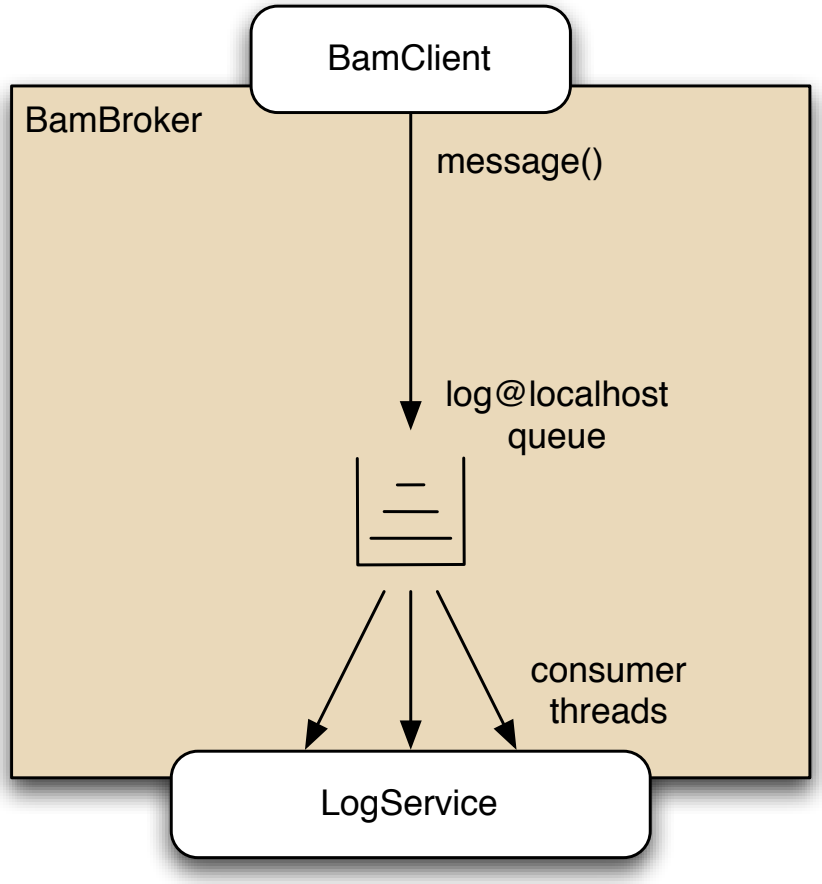
administration consoles. In out example, a servlet sends a message to an internal logging service.

The sending agent calls `message()` to send a message with the jid address of the target agent (to), and a message payload. The broker routes the message based on the address and calls the `message()` method on the target service to process the message. As the service processes the message, the client returns from its `message()` call and continues processing. If a new message arrives for the service, the broker will queue the message until the service is ready to process it. The queue isolates the producing thread from the service, improving response time and even allowing to be processed on a separate machine or cluster as a batch job.



The message can be any serializable object appropriate to the service, either a custom Java model bean, or an XML string, or a defined protocol message, like a Jabber IM message. This flexibility lets services define messages appropriate to the application, and avoids tying the service into knots trying to conform to a restricted encoding like SOAP. If remoting is used, the remoting protocol might restrict the possible messages. HMTP (Hessian) will allow any serializable object, while XMPP (Jabber) is restricted to XML and `ImMessage` .

Since BAM messages are addressible and routed through the broker, clients have flexibility in choosing their destination at runtime. BAM messages use a JID (Jabber ID) for addressing, which looks like `service@domain` or `service@domain/resource` . The second `service@domain/resource` is used for dynamic agent, e.g. a user logged into messaging with a cellphone.



Writing a BAM client involves the following steps:

1. Create a `BamClient`

2. Sending messages

In the example, the servlet creates a `BamClient` and sends the message. The special jid syntax "service@" uses the local domain as a destination.

```
package example;

import javax.servlet.*;
import com.caucho.bam.BamClient;

public class TestClient extends GenericServlet
{
  private BamClient _client = new BamClient();

  public void service(ServletRequest req, ServletResponse response)
  {
    _client.message("test@", "Hello, world!");
  }
}
```

Example: TestClient.java

Writing a BAM message/queuing service involves the following steps:

1. Implementing `BamService` (usually by extending `SimpleBamService`)
2. Configuring the `BamService` using <`bam-service`> , which will automatically register the service with the `BamBroker` .
3. Receiving messages by overriding `BamStream` methods in `SimpleBamService` .

By configuring <`bam-service`> , the service automatically gains a queuing ability. The broker will queue the message and spawn a new thread before calling the service's `message` , in order to isolate the receiver from the sender. Advanced applications can disable the queue if appropriate.

```
package example;

import com.caucho.bam.SimpleBamService;
import java.io.Serializable;
import java.util.logging.*;

public class LogService extends SimpleBamService
{
  private static final Logger log
    = Logger.getLogger(LogService.class.getName());

  @Override
  public void message(String to, String from, Serializable value)
  {
    log.info(this + " message from=" + from + " value=" + value);
  }
}
```

Example: LogService.java

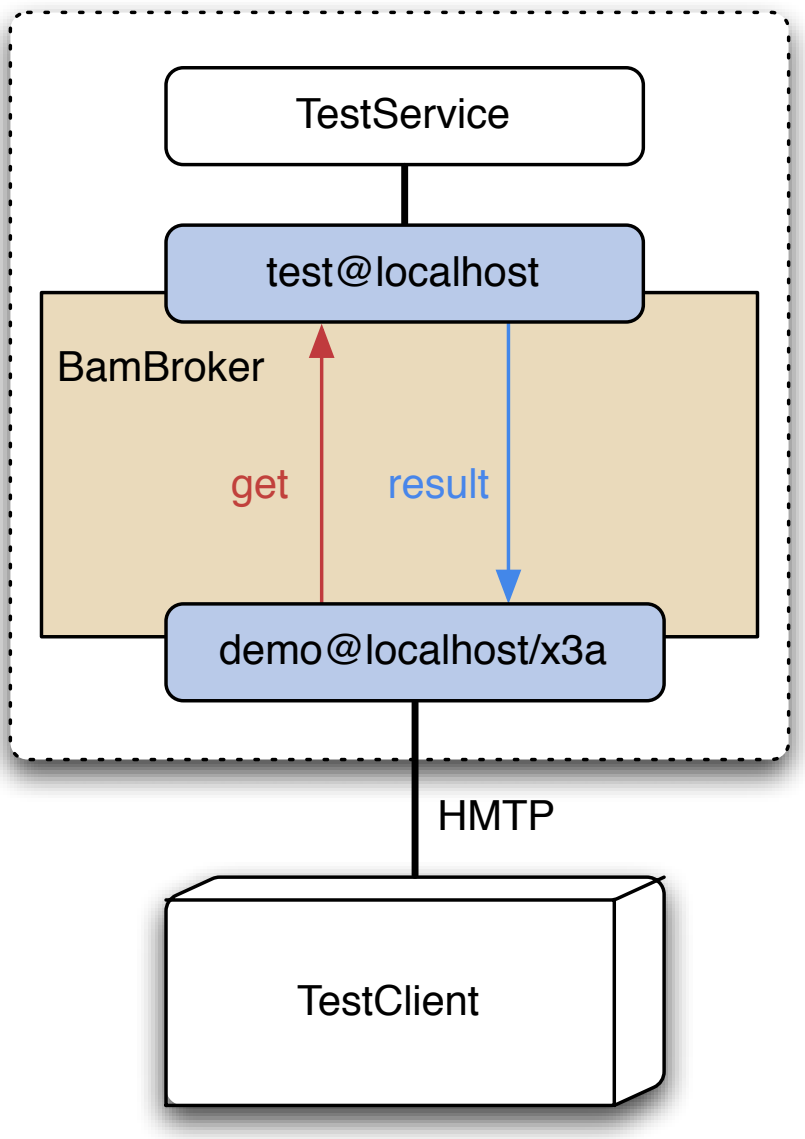
The BAM configuration the service configured with Resin IoC.

```
<web-app xmlns="http://caucho.com/ns/resin">

    <bam-service name="test" class="example.LogService"/>

</web-app>
```

Example: WEB-INF/resin-web.xml

Client queryGet (RPC) example

Remote calls in BAM can query or update a service based on the type of the query message. Since the query is typed, a service can be defined by the set of query types it understands and even mixin multiple capabilities, like implementing both a chat and a pub/sub service. In this example, we just query a service for some basic information.



We'll use a remote client to show how BAM can be used as a remote service as well a local organization. `HmtpClient` , which extends the same `BamConnection` API as the local `BamClient` implements BAM using Hessian as the wire protocol. For local messages, we could use `BamClient` instead. Once the connection is established, the remaining code is identical.

When you create a `HmtpClient` , you'll send it the URL of the HMTP service, then call `connect()` and `login` to authenticate. The `login()` method will

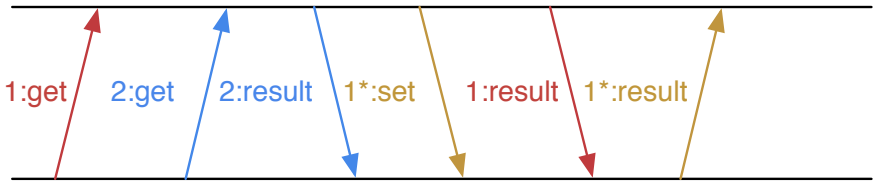
register an agent with the broker, letting the client send and receive messages. The example sends a single `TestQuery` query to the `test@localhost` service.

```
package example;

import com.caucho.hmtp.client.HmtpClient;

public class TestClient
{
  public static void main(String []args)
    throws Exception
  {
    HmtpClient client = new HmtpClient("http://localhost:8080/hmtp");
    client.connect();
    client.login("user@localhost", null);

    Object value = client.queryGet("test@localhost", new TestQuery());

    System.out.println(value);

    client.close();
  }
}
```

Example: TestClient.java

To implement the server side of an RPC call, the service implements `queryGet` or `querySet` and examines the query to see if it understands the query class. To simplify the query dispatching, `SimpleBamService` introspects the methods looking for `@QueryGet` annotations and creating a map of query types, in this case `TestQuery` .

For a query, the service must always send a QueryResult message or a QueryError message with the same `id` back to the caller to match responses to the calls. If the service understands the query, it will send a result message and return true. If it does not understand the query, it will return false, which tells the broker to send a query error message.



The `id` matches responses to the corresponding queries. Since BAM is a bidirectional streaming architecture, queries can be unordered and start from either direction. The `id` turns this unordered mess into a coherent request-response pattern for RPC-style calls.

```
package example;

import com.caucho.bam.SimpleBamService;
import com.caucho.bam.annotation.QueryGet;

public class TestService extends SimpleBamService
{
  @QueryGet
  public boolean testQueryGet(long id, String to, String from,
                              TestQuery query)
  {
    getBrokerStream().sendQueryResult(id, to, from, "hello response");

    return true;
  }
}
```

Example: TestService.java

The configuration for a service now has two components:

1. Any registered `BamService` , e.g. the TestService
2. The exposed HMTP service protocol, implemented with `HempServlet`

The BAM service itself does not know or care that it's being called remotely. The remote HMTP servlet exists only so the remote client can login to the local broker.

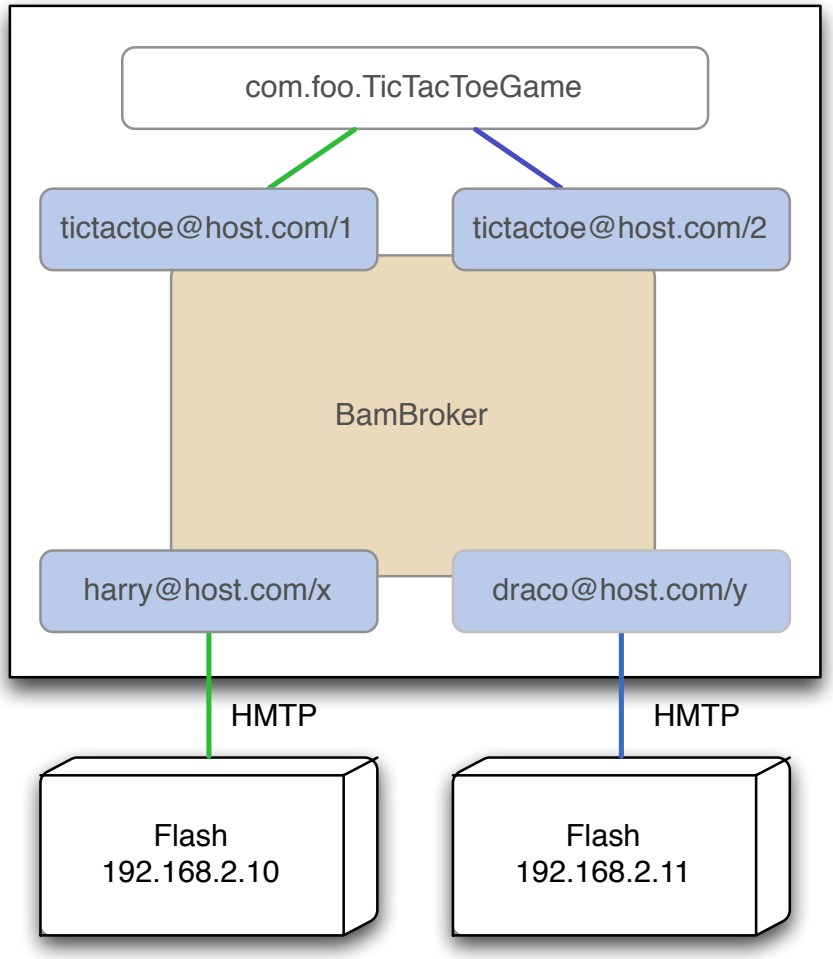
```
<web-app xmlns="http://caucho.com/ns/resin">

    <bam-service name="test" class="example.TestService"/>

    <servlet-mapping url-pattern="/hmtp"
                     servlet-class="com.caucho.hemp.servlet.HempServlet"/>

</web-app>
```

Example: WEB-INF/resin-web.xml

Brokered Agent Messaging (BAM)

Applications using BAM will generally follow a Brokered Agent Messaging pattern, a hub-and-spoke messaging topology where the agents act as dynamic services, joining and detaching from the broker as the application progresses.

Services and clients register one or more agents with the `BamBroker` and send messages between the agents. Each remote client will register a local agent with the local broker. and each service will register one or more agents with the broker. In a tic-tac -toe game, the game instance might register two agents: one for each player in a particular game.

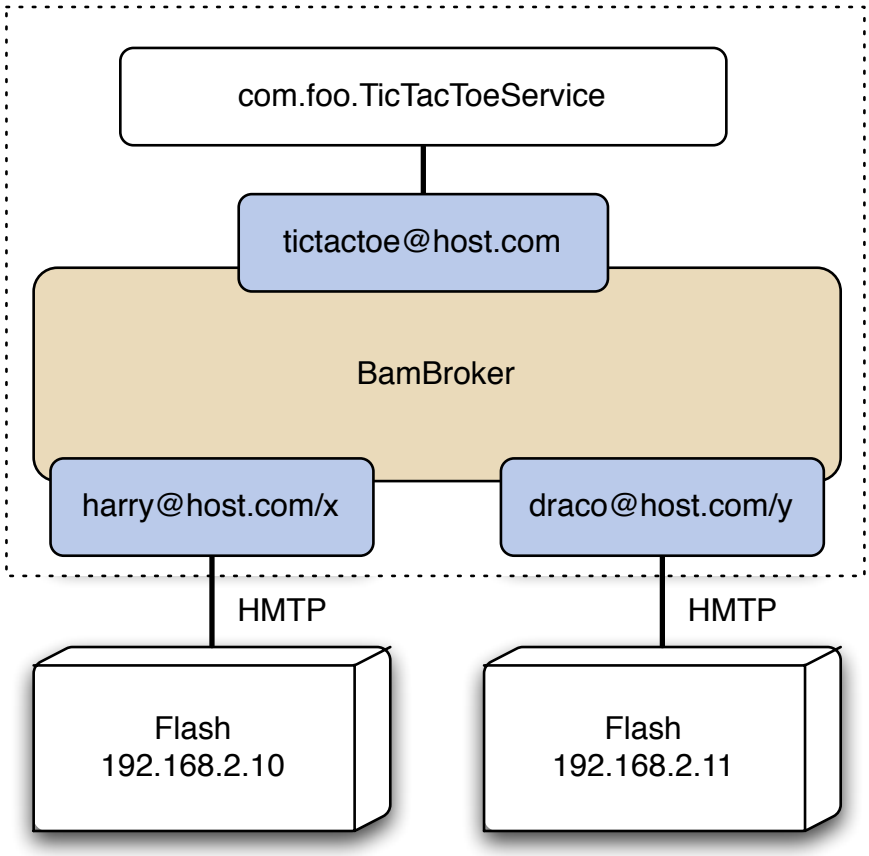


The diagram above has four agents: two agents for the game's players, and one agent for each logged-in user. `tictactoe@host.com/1` is the game's agent for player #1, and `harry@host.com/x` is Harry's agent for his flash client. In the tic-tac-toe game, each user's agent talks to the matching game player, so `harry@host.com/x` always talks to `tictactoe@host.com/1` , and `draco@host.com/y` always talks to `tictactoe@host.com/1` .

The game's agents are ephemeral. When a new game begins, a `TicTacTocGame` instance registers two new agents for the new game, with unique names, e.g. `tictactoe@host.com/3` and `tictactoe@host.com/4` . When the game ends, the instance will unregister its agents.

Because the game's agents are only created when a game begins, the tic-tac-toe game has a persistent agent for registration, `tictactoe@host.com` . When Harry logs on, the client will send a query to `tictactoe@host.com` asking for

a new game. As soon as Draco asks for a match, the registration server will create a new game instance and tell Harry's client the name of his player agent, `tictactoe@host.com/1` .



Addressing (JIDs)

BAM resources all have unique identifiers called JIDs (Jabber IDs), which look and act like extended email addresses. Because IM applications can have multiple connections for the same user, each address has an optional resource providing a unique name for the connection.

The id looks like:

```
\texttt{service} @ \texttt{domain} / \texttt{resource}
```

- `domain` is an virtual host name. Like email or HTTP, BAM messages can be routed to any internet host, i.e. BAM is a federated architecture, not a strict client-server architecture.

- `service` is the service name within the domain. In IM services, each user is represented as a separate service.
- `resource` is an ephemeral agent name. Since each agent needs a addressable name, the resource identifies each user login or service agent uniquely.

The `service` and `resource` are optional.

JID	DESCRIPTION
ferg@foo.com	IM user resource
ferg@foo.com/xB8	User login agent, i.e. the BAM address corresponding to a logged in IM session.
batch@foo.com	Application queuing service (like an EJB message driven bean)
mail@foo.com	Mail notification service
tictactoe@foo.com	tic-tac-toc game manager resource
tictactoe@foo.com/1	player #1 agent of a tic-tac-toe game
tictactoe@foo.com/2	player #2 agent of a tic-tac-toe game
tictactoe@foo.com/3	player #1 agent of a tic-tac-toe game #2
tictactoe@foo.com/4	player #2 agent of a tic-tac-toe game #2
myroom@foo.com	chatroom instance
myroom@foo.com/harry	chatroom nickname for user #1
myroom@foo.com/draco	chatroom nickname for user #2
announcements@foo.com	publish/subscribe resource

example jids

bam-service

<bam-service> configures a `BamService` to listen for messages. The service is always a single multithreaded instance like a servlet; it is never pooled.

<bam-service> can be used as a simple queueing service, replacing JMS queues and ejb-message-bean. By default the queue is consumed with a single thread. Unlike EJB message driven beans, <bam-service> uses a single instance like a servlet, not a pool of instances.

The bean has full access to Resin-IoC capabilities, including dependency injection, transaction attributes, and aspect interception.

ATTRIBUTE	DESCRIPTION	DEFAULT
class	Classname of the BamService bean	required
init	IoC configuration for the listener bean	

name	The JID of the service, used by clients to send messages to	
thread-max	The number of threads to handle the queue. If 0, the service is not queued, but handles messages directly.	1
uri	A shortcut name for the service, defined in META-INF/caucho/com.caucho.	Service.

<bam-service> attributes

```
element ejb-message-bean {
  (class | uri)
  & init?
  & name?
  & thread-max?
}
```

Resin Services

Resin includes a number of predefined BAM services for JMS compatibility, logging, and mail messages. The services are all configured with <bam-service> .

URI	DESCRIPTION
caucho.jms	forwards messages to a JMS queue
caucho.log	logs messages to java.util.logging
caucho.mail	sends an email with a summary of recent messages
caucho.php	configures a service written in PHP

Predefined Services

JMS

The JMS service forwards BAM messages to a JMS queue, wrapping each message in a JMS ObjectMessage. Applications can use this BAM to JMS bridge to queue messages from a Java client.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-connection-factory name="jms_cf" uri="resin:"/>
  <jms-queue name="queue" uri="memory:"/>

  <bam-service name="jms" uri="caucho.jms:">
    <init>
      <connection-factory>${jms_cf}</connection-factory>
      <queue>${queue}</queue>
    </init>
  </bam-service>

</web-app>
```

Example: JMS configuration

Logging

The logging service adds BAM messages to a java.util.logging Logger. An application could use the logging service as a chat-room recording or debugging service.

ATTRIBUTE	DESCRIPTION
name	logger name
level	logger level

caucho.log attributes

```
<web-app xmlns="http://caucho.com/ns/resin">

  <bam-service name="log" uri="caucho.log:">
    <init>
      <name>com.foo.chat</name>
      <level>info</level>
    </init>
  </bam-service>

</web-app>
```

Example: Log configuration

Mail

The mail service sends BAM messages to an email address. This can be used to notify any issues with an application that may need administration.

ATTRIBUTE	DESCRIPTION

delay-time	a pause interval to gather groups of messages before sending
mail-session	a javax.mail.Session object
properties	javamail properties
subject	the mail subject
to	a mail destination address

caucho.mail attributes

```
<web-app xmlns="http://caucho.com/ns/resin">

  <bam-service name="mail" uri="caucho.mail:">
    <init>
      <to>bam@foo.com</name>
      <subject>BAM Notification</subject>
      <properties>
        mail.from=bamservice@foo.com
      </properties>
    </init>
  </bam-service>

</web-app>
```

Example: Mail configuration

PHP

The php service configures a PHP script as a service handler.

ATTRIBUTE	DESCRIPTION
script	a path to the PHP script

caucho.php attributes

```
<web-app xmlns="http://caucho.com/ns/resin">

  <bam-service name="php" uri="caucho.php:">
    <init>
      <script>WEB-INF/php/php-service.php</script>
    </init>
  </bam-service>

</web-app>
```

Example: PHP configuration

```
<?php

bam_message($to, $from, $message)
{
  resin_debug($message);
}

bam_dispatch();

?>
```

Example: WEB-INF/php/php-service.php

Protocols

Local (JVM calls)

Local JVM applications can use BAM to organize internal applications like queuing consumers to replace JMS, or service-oriented architectures. Because BAM passes messages by reference, it avoids the cpu and memory overhead of serialization, improving performance.

The local clients can take advantage of BAM's federated addressing and can send messages to foreign machines while logging into the local broker.

HMTP (Hessian)

HMTP (Hessian Message Transport Protocol) is a streaming mode of the Hessian protocol supporting BAM. Since Hessian can serialize any Java object, it can support all of BAM's capabilities for remote services and since Hessian has been ported to many languages including Flash/ActionScript, multilanguage clients can use BAM directly though HMTP.

XMPP (Jabber)

BAM is an adaptation of the XMPP (Jabber) instant messaging protocol. Where XMPP (Xml Messaging and Presence Protocol) is based on XML, HMTP (Hessian Message Transport Protocol) is based on Hessian. Because BAM is designed to follow XMPP, its architecture and protocols are essentially identical until the very lowest layer.

Because of the close relationship to XMPP, you may want to browse the XMPP specifications for a deeper understanding of how HMTP works. Since XMPP is only a wire protocol, not an API, it does not include all of the HMTP classes, but the architecture remains the same.

The primary advantages HMTP offers over XMPP include the performance advantages of Hessian over XML, and more importantly a more strict layering than XMPP provides. Because the payloads of the HMTP messages are all `Serializable` , applications have enormous flexibility in developing their own

messages using application objects. In contrast, XMPP messages are always XML, so applications are not only restricted to XML data, but also must create their own XML parsers and formatters.

Packet types

BAM provides three categories of packets: messages, queries (rpc), and presence announcements. A queuing or messaging application might only use message packets, while a pub/sub service might use messages and queries. Chat, conference room, and monitoring software will use presence messages to announce joining or startup events.

Messages are unidirectional fire-and-forget packets. They can be used for queuing systems as a replacement JMS queues.

Queries are request-response pairs. Each request must have a corresponding response or error.

Presence announcements are used to organize subscriptions. There are presence announcements to subscribe and unsubscribe, and presence notifications that a user has logged on, sent to all other users subscribed to his presence.

Message Packets

The main Message packet contains a target ("to"), a sender ("from"), and a payload ("value"). In BAM, the payload can be any serializable value. Example messages could be IM text messages, queued tasks, administration console graph, game updates, or updated stock quotes. Since BAM is bidirectional, messages can flow to and from any client.

- Message - sends a message to a resource
- MessageError - sends a message error to a resource

Query Packets

Query packages are RPC get and set packets with a matching response or error. Because the query will always have a matching response packet or an error packet, clients can either block for the result or attach a callback.

Like the other packets, queries are bidirectional, so a service can query a client as well as the usual client querying the server.

Query packets have an associated `id` field to match requests with responses. The client will increment the `id` for each new query.

- QueryGet - sends an information request
- QuerySet - sends an action query
- QueryResponse - returns a response
- QueryError - returns an error

Presence Packets

Presence packets send specialized information for subscription notification. Many applications will not need to use any presence packets at all.

- Presense - sends a presence (login) notification
- PresenseUnavailable - sends unavailable (logout) notification
- PresenseProbe - query probe for IM clients
- PresenseSubscribe - request to subscribe to a service
- PresenseSubscribed - acknowledgement of a subscription
- PresenseUnsubscribe - notification of an unsubscription
- PresenseUnsubscribed - notification of an unsubscription
- PresenseError - error message

API

Applications use `BamClient` , `BamStream` and `SimpleBamService` as their main BAM APIs. Conceptually, BAM is designed around `BamStream` as the central, unidirectional streaming interface. `BamClient` and `SimpleBamService` provide appropriate facades around the underlying streaming architecture for a cleaner and simpler application model.

- `BamBroker` - Interface for the hub message routing, and service/agent registration.
- `BamClient` - Client class for sending and receiving messages calls from a local Java client, implementing `BamConnection` .
- `BamConnection` - Client interface for sending and receving messages/rpc calls.
- `BamConnectionFactory` - Factory interface for creating connections to a broker.
- `BamQueryCallback` - client API to handle asyncronous RPC responses.
- `BamService` - Interface implemented by all registered BAM services.
- `BamServiceManager` - Optional interface for dynamically registered services, e.g. an IM user manager.
- `BamStream` - Stream interface for all message routing and filtering.
- `HmtpClient` - client class for remote Hessian messaging
- `SimpleBamService` - abstract class extended by most BAM services, providing automatic typed dispatching.
- `XmppClient` - client class for remote XMPP (Jabber)

Client API

BamClient `BamClient` is the primary client class for local clients. The `BamClient` will automatically create a connection with the local broker. Messages are sent using the `BamConnection` methods. Messages are received by setting a BamStream handler.

```
package com.caucho.bam;

public class BamClient implements BamConnection
{
  public BamClient();
  public BamClient(String uid, String password);
  public BamClient(String uid, String password, String resource);

  String getJid();

  // message
  void message(String to, Serializable value);

  // rpc
  Serializable queryGet(String to, Serializable query);
  Serializable querySet(String to, Serializable query);
  void queryGet(String to, Serializable query, BamQueryCallback callback);
  void querySet(String to, Serializable query, BamQueryCallback callback);

  // presence
  void presence(Serializable data);
  void presence(String to, Serializable data);
  void presenceUnavailable(Serializable data);
  void presenceUnavailable(String to, Serializable data);
  void presenceProbe(String to, Serializable data);
  void presenceSubscribe(String to, Serializable data);
  void presenceSubscribed(String to, Serializable data);
  void presenceUnsubscribe(String to, Serializable data);
  void presenceUnsubscribed(String to, Serializable data);
  void presenceError(String to, Serializable data, BamError error);

  // callback handler for receiving messages
  void setStreamHandler(BamStream handler);

  // raw stream to return rpc responses
  BamStream getBrokerStream();
}
```

BamConnection `BamConnection` is the primary client interface for both local and remote clients. Messages are sent using the `BamConnection` methods. Messages are received by setting a BamStream handler.

An active `BamConnection` has an associated agent registered with the broker. The agent's jid is available with the `getJid()` call.

For clients that need low-level access to the broker stream, e.g. to implement an RPC/Query handler, `getBrokerStream()` returns the underlying stream.

```
package com.caucho.bam;

public interface BamConnection
{
  String getJid();

  boolean isClosed();
  void close();

  // message
  void message(String to, Serializable value);

  // rpc
  Serializable queryGet(String to, Serializable query);
  Serializable querySet(String to, Serializable query);
  void queryGet(String to, Serializable query, BamQueryCallback callback);
  void querySet(String to, Serializable query, BamQueryCallback callback);

  // presence
  void presence(Serializable data);
  void presence(String to, Serializable data);
  void presenceUnavailable(Serializable data);
  void presenceUnavailable(String to, Serializable data);
  void presenceProbe(String to, Serializable data);
  void presenceSubscribe(String to, Serializable data);
  void presenceSubscribed(String to, Serializable data);
  void presenceUnsubscribe(String to, Serializable data);
  void presenceUnsubscribed(String to, Serializable data);
  void presenceError(String to, Serializable data, BamError error);

  // callback handler for receiving messages
  void setStreamHandler(BamStream handler);

  // raw stream to return rpc responses
  BamStream getBrokerStream();
}
```

BamConnectionFactory The `BamConnectionFactory` produces `BamConnection` agents for client code. Typically, the factory implementation will be a `BamBroker` , although that is not required by the clients.

```
package com.caucho.bam;

public interface BamConnectionFactory
{
  BamConnection getConnection(String uid, String password);

  BamConnection getConnection(String uid, String password, String resourceId);
}
```

BamQueryCallback `BamQueryCallback` is used for callback-style RPC. When the query response completes, the agent will call the `BamQueryCallback`

with the query's response.

```
package com.caucho.bam;

public interface BamQueryCallback
{
  void onQueryResult(String to, String from, Serializable value);

  void onQueryError(String to, String from, Serializable value,
                    BamError error);
}
```

Remote Client API

`HmtpClient` is the remote client API for Java clients. Most of the methods are extended from `BamConnection` . The additional method provide some control for connection and login. Once the client is logged in, applications will typically use `BamConnection` methods to send messages and set handlers to receive messages.

HmtpClient

```
package com.caucho.bam;

public class HmtpClient implements BamConnection
{
  public HmtpClient(String url);

  public void connect() throws IOException;

  public void login(String uid, String password);

  // BamConnection methods
  String getJid();

  boolean isClosed();
  void close();

  void setStreamHandler(BamStream handler);

  void message(String to, Serializable value);

  Serializable queryGet(String to, Serializable query);
  Serializable querySet(String to, Serializable query);

  void queryGet(String to, Serializable query, BamQueryCallback callback);
  void querySet(String to, Serializable query, BamQueryCallback callback);

  void presence(Serializable data);
  void presence(String to, Serializable data);
  void presenceUnavailable(Serializable data);
  void presenceUnavailable(String to, Serializable data);
  void presenceProbe(String to, Serializable data);
  void presenceSubscribe(String to, Serializable data);
  void presenceSubscribed(String to, Serializable data);
  void presenceUnsubscribe(String to, Serializable data);
  void presenceUnsubscribed(String to, Serializable data);
  void presenceError(String to, Serializable data, BamError error);

  BamStream getBrokerStream();
}
```

Protocol(Packet) API

BamStream `BamStream` is the core streaming API for the broker and its registered agents. It is simply a combination of all the message, query and presence packets.

Applications will implement `HmtpQueryStream` to receive RPC calls and responses from the agent. If the application implements `sendQueryGet` , it must either send a `QueryResponse` to the sender, or send a `QueryError` or return false from the method. Queries will always have a response or an error.

The presence methods implement the specialized subscription and presence messages. IM applications use presence messages to announce availability to people in a buddy list (roster).

Publish/Subscribe applications can also use subscription packets to subscribe and unsubscribe from the publishing service.

```
package com.caucho.bam;

public interface BamStream
{
  public String getJid();

  // message
  public void message(String to, String from, Serializable value);
  public void messageError(String to, String from, Serializable value,
                           BamError error);

  // rpc
  boolean queryGet(long id, String to, String from, Serializable query);
  boolean querySet(long id, String to, String from, Serializable query);
  void queryResult(long id, String to, String from, Serializable value);
  void queryError(long id, String to, String from, Serializable query,
                  BamError error);

  // presence
  void presence(String to, String from, Serializable data);
  void presenceUnavailable(String to, String from, Serializable data);
  void presenceProbe(String to, String from, Serializable data);
  void presenceSubscribe(String to, String from, Serializable data);
  void presenceSubscribed(String to, String from, Serializable data);
  void presenceUnsubscribe(String to, String from, Serializable data);
  void presenceUnsubscribed(String to, String from, Serializable data);
  void presenceError(String to, String from, Serializable data,
                     BamError error);
}
```

Service APIs

BamBroker `BamBroker` is the central player in the HMTP server. It's responsible for routing messages between the agents, for any forwarding to remote servers, and managing dynamic agents and services.

For all that responsibility, the API is fairly simple. The `BamBroker` extends `BamConnectionFactory` , enabling client agents, and allows custom `BamService` services to be implemented. Most importantly, it implements a broker stream (`BamStream`) which serves as the destination for all inbound messages.

```
package com.caucho.bam;

public interface BamBroker extends BamConnectionFactory
{
  BamStream getBrokerStream();

  void addService(BamService service);
  void removeService(BamService service);

  void addServiceManager(ServiceManager manager);
}
```

BamService `BamService` represents a registered, persistent service with a known jid address. Typically the services will be registered in a configuration file, although they can also be created dynamically using the `BamServiceManager` . Most applications will extend the `SimpleBamService` instead of implementing `BamService` directly.

The key methods are `getJid` and `getAgentStream` . The jid is used for registration with the `BamBroker` and `getAgentStream` is used to receive any messages.

The additional methods are used for specialized applications like instant messaging and multiuser-chat, to manage clients logging in.

```
package com.caucho.bam;

public interface BamService
{
  public String getJid();

  public BamStream getAgentStream();

  public boolean startAgent(String jid);
  public boolean stopAgent(String jid);

  public void onAgentStart(String jid);
  public void onAgentStop(String jid);

  public BamStream getAgentFilter(BamStream stream);
  public BamStream getBrokerFilter(BamStream stream);
}
```

BamServiceManager `BamServiceManager` is a specialized manager for finding persistent sessions. In instant messaging, for example, the registered users might be stored in a database. When a message goes to harry@host.com, the `BamServiceManager` will lookup the appropriate user.

```
package com.caucho.bam;

public interface BamServiceManager
{
  public BamService findService(String jid);
}
```

SimpleBamService

```
package com.caucho.bam;

abstract public class SimpleBamService implements BamService, BamStream
{
  public String getJid();

  // message
  public void message(String to, String from, Serializable value);
  public void messageError(String to, String from, Serializable value,
                           BamError error);

  // rpc
  boolean queryGet(long id, String to, String from, Serializable query);
  boolean querySet(long id, String to, String from, Serializable query);
  void queryResult(long id, String to, String from, Serializable value);
  void queryError(long id, String to, String from, Serializable query,
                  BamError error);

  // presence
  void presence(String to, String from, Serializable data);
  void presenceUnavailable(String to, String from, Serializable data);
  void presenceProbe(String to, String from, Serializable data);
  void presenceSubscribe(String to, String from, Serializable data);
  void presenceSubscribed(String to, String from, Serializable data);
  void presenceUnsubscribe(String to, String from, Serializable data);
  void presenceUnsubscribed(String to, String from, Serializable data);
  void presenceError(String to, String from, Serializable data,
                     BamError error);

  // BamService methods
  public BamStream getAgentStream();

  public boolean startAgent(String jid);
  public boolean stopAgent(String jid);

  public void onAgentStart(String jid);
  public void onAgentStop(String jid);

  public BamStream getAgentFilter(BamStream stream);
  public BamStream getBrokerFilter(BamStream stream);
}
```

annotations

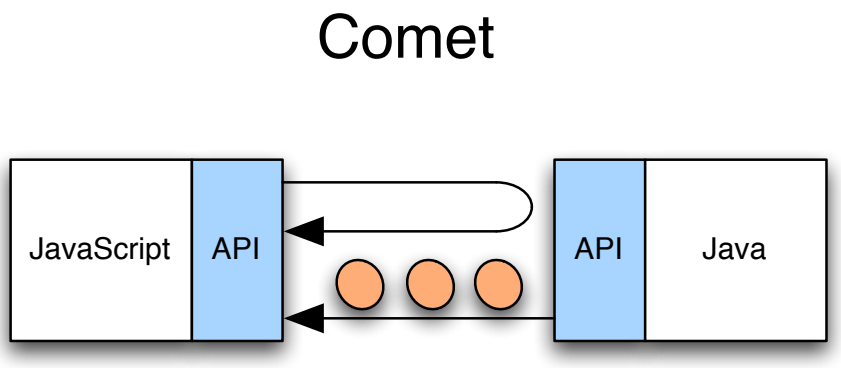
```
package com.caucho.bam.annotation;

public @interface Message {}
public @interface MessageError {}

public @interface QueryGet {}
public @interface QuerySet {}
public @interface QueryResult {}
public @interface QueryError {}

public @interface Presence {}
public @interface PresenceProbe {}
public @interface PresenceUnavailable {}
public @interface PresenceSubscribe {}
public @interface PresenceSubscribed {}
public @interface PresenceUnsubscribe {}
public @interface PresenceUnsubscribed {}
public @interface PresenceError {}
```

Chapter 20

Comet

20.1 Comet/Server-Push Servlet

Resin's server-push (Comet) API lets server application push new data to the client as it becomes available. Administration and monitoring applications need to continually update the client when new information becomes available to the server.



The architecture in the picture uses two HTTP streams to the server application, one for normal client-server requests, and a second unidirectional stream to send updates from the server to the client. In this example, we're using a browser with JavaScript, but the same architecture applies to a more sophisticated Flash monitoring application sending Hessian packets to update the monitoring display.

Example Overview

The example updates a comet.html page every two seconds with new data. In this case, just an updated counter.

The components of the Comet/AJAX application look like:

- Protocol: JavaScript function calls with a trivial argument.
- Client:
  - View: HTML updated by JavaScript AJAX
  - Controller: call server with an <iframe>
- Server:
  - Service: TimerService manages the comet connections and wakes them with new data.
  - Servlet: TestCometServlet generates <script> protocol tags from new data from the TimerService on each `resume` .
  - State: CometState encapsulates both the item's state (the timer count), and the CometController needed to wake the servlet and pass updated data.

Streaming Protocol: <script> tags

The comet HTTP stream is a sequence of <script> tags containing JavaScript commands to update the browser's display. Because the browser executes the script as part of its progressive rendering, the user will see the updates immediately without waiting for the entire HTTP request to complete.

In our example, the packet is a JavaScript `comet_update(data)` call, which updates the text field with new data. Here's an example of the packet stream:

```
<script type="text/javascript">
window.parent.comet_update(1);
</script>

<!-- 2 second delay -->

<script type="text/javascript">
window.parent.comet_update(2);
</script>

<!-- 2 second delay -->

<script type="text/javascript">
window.parent.comet_update(3);
</script>
```

Update JavaScript packets

More sophisticated comet applications will use a dynamic-typed protocol to update the client. Browser-based applications could use JSON to update the client and Flash-based applications might use Hessian. In all cases, the protocol

must be kept simple and designed for the client's requirements. Design separate, simple protocols for Flash and JavaScript browsers, rather than trying to create some complicated general protocol.

Browser Client

The JavaScript command stream updates a parent HTML file which defines the JavaScript commands and launches the Comet servlet request with an <iframe> tag. Our `comet_update` function finds the HTML tag with `id="content"` and updates its HTML content with the new data from the server.

```
<html>
<body>

Server Data:
<span id="content">server data will be shown here</span>

<script type="text/javascript">
function comet_update(value) {
  document.getElementById('content').innerHTML = value;
};
</script>

<iframe src="comet"
        style="width:1px;height:1px;position:absolute;top:-1000px"></iframe>

</body>
</html>
```

comet.html

CometController

The CometController is Resin's thread-safe encapsulation of control and communication from the application's service to the Comet servlet. Applications may safely pass the `CometController` to different threads, wake the servlet with `wake()` , and send data with `setAttribute` .

In the example, the `TimerService` passes the updated count to the servlet by calling `setAttribute("caucho.count", count)` , and wakes the servlet by calling `wake()` . When the servlet resumes, it will retrieve the count using `request.getAttribute("caucho.count")` . Note, applications must only use the thread-safe `CometController` in other threads. As with other servlets, the `ServletRequest` , `ServletResponse` , writers and output stream can only be used by the servlet thread itself, never by any other threads.

```
package com.caucho.servlet.comet;

public interface CometController
{
  public void wake();

  public Object getAttribute(String name);
  public void setAttribute(String name, Object value);
  public void removeAttribute(String name);

  public void close();
}
```

com.caucho.servlet.comet.CometController

Comet Servlet

The comet servlet has three major responsibilities:

1. Process the initial request (`service`).

2. Register the `CometController` with the service (`service`).

3. Send streaming data as it becomes available (`resume`).

Like other servlets, only the comet servlet may use the `ServletRequest` , `ServletResponse` or any output writer or stream. No other thread may use these servlet objects, and the application must never store these objects in fields or objects accessible by other threads. Even in a comet servlet, the servlet objects are not thread-safe. Other services and threads must use the `CometController` to communicate with the servlet.

Process the initial request: our servlet just calls `setContentType("text/html")` , since it's a trivial example. A real application would do necessary database lookups and possibly send more complicated data to the client.

Register the `CometController` : our servlet registers the controller with the timer service by calling `addCometState` . In general, the application state object will contain the `CometController` as part of the registration process.

Send streaming data:. The `TimerService` will set new data in the `"comet.count"` attribute and `wake()` the controller. When the servlet executes the `resume()` method, it will retrieve the data, and send the next packet to the client.

```
package example;

import java.io.*;

import javax.servlet.http.*;
import javax.servlet.*;

import com.caucho.servlet.comet.GenericCometServlet;
import com.caucho.servlet.comet.CometController;

public class TestComet extends GenericCometServlet {
  @Override
  public boolean service(ServletRequest request,
                         ServletResponse response,
                         CometController controller)
    throws IOException, ServletException
  {
    HttpServletRequest req = (HttpServletRequest) request;
    HttpServletResponse res = (HttpServletResponse) response;

    res.setContentType("text/html");

    TestState state = new TestState(controller);

    _service.addCometState(state);

    return true;
  }

  @Override
  public boolean resume(ServletRequest request,
                        ServletResponse response,
                        CometController controller)
    throws IOException, ServletException
  {
    HttpServletRequest req = (HttpServletRequest) request;
    HttpServletResponse res = (HttpServletResponse) response;

    PrintWriter out = res.getWriter();

    String count = req.getAttribute("comet.count");

    out.print("<script type='text/javascript'>");
    out.print("comet_update(" + count + ");");
    out.print("</script>");

    return true;
  }
}
```

example/TestCometServlet.java

The connection can close for a number of reasons. Either the `service()` or `resume()` methods may return false, telling Resin to close the connection. The

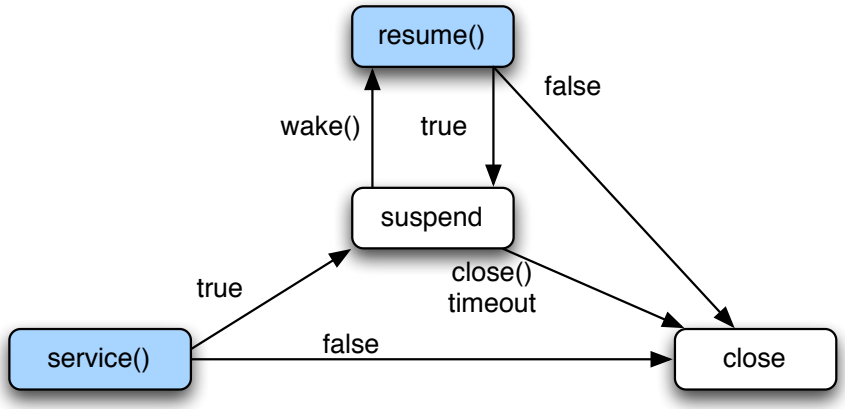
service might call `CometController.close()` which will also close the connection. Finally, the client may close the connection itself.

The sequence of calls for the example looks like the following:

1. `servlet.service()` is called for the initial request
2. `_service.addCometState()` registers with the `TimerService`
3. after the `service()` completes, Resin suspends the servlet.
4. The `TimerService` detects an event, in this case the timer event.
5. The `TimerService` calls `controller.setAttribute()` to send new data.
6. The `TimerService` calls `controller.wake()` to wake the servlet.
7. `servlet.resume()` processes the data and sends the next packet.
8. After the `resume()` completes, Resin suspends the servlet again and we repeat as after step #3.
9. After the 10th data, the `TimerService` calls `controller.close()` , closing the servlet connection.

Comet Servlet State Machine

The sequence of comet servlet calls looks like the following state machine. After the initial request, the servlet spends most of its time suspended, waiting for the `TimerService` to call `wake()` .



Chapter 21

Remoting

21.1 Resin Remoting

Example: Hello, World

The Hello, World example shows the primary steps involved in creating a Resin remoting service:

1. Creating the public API
2. Creating the Java implementation
3. Configuring the service in the resin-web.xml and choosing the protocol
4. Configuring and using the client

API - defining the protocol

Defining the protocol cleanly and clearly is the most critical aspect of designing a remote service. Because you will be sharing the protocol API among many other developers using different languages, it's critical to create a good design.

The API classes are used for both the client and the server, ensuring compatibility. Even when the clients are expected to be written in a different language, e.g. Flash, C# or JavaScript, the API serves both as documentation and validation of the protocol. In the case of C#, the API can be translated automatically using reflection for strict compile-time validation.

In this example, the API is easy. It's just a single method call `hello` returning a greeting string.

```
package qa;

public interface Hello {
  public String hello();
}
```

Example: Hello.java - API

Service - implementing the service

The service implementation is a plain Java object that can optionally use Resin-IoC capabilities for dependency injection. The service is multithreaded, so it's the service-developer's responsibility to handle any synchronization or transaction issues, just like writing a servlet.

In this example, the implementation is trivial, just returning the "hello, world" string. More complicated services might delegate to WebBeans or EJB services.

```
package qa;

public class MyService implements Hello {
  public String hello()
  {
    return "hello, world";
  }
}
```

Example: MyService.java - Service

Configuration - exporting the protocol

Web-based remoting protocols are exported using the HTTP protocol with well-known URLs; they're essentially fancy servlets. Resin lets you configure your services just like a servlet. The only additional configuration necessary is choosing the protocol.

Protocol drivers like Hessian or CXF register the protocol implementation with a URI scheme like "hessian:" or "cxf:". Your service configuration will just select the appropriate protocol in a <protocol> configuration tag.

In the example, we'll use Hessian, since it's a fast binary protocol with several language implementations. If you want to export multiple protocol bindings, you can just add new <servlet-mapping> definitions.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <servlet-mapping url-pattern="/hello" servlet-class="qa.MyService">
    <protocol uri="hessian:"/>
  </servlet-mapping>

</web-app>
```

Example: WEB-INF/resin-web.xml - Hessian Service

We can easily change the protocol to use CXF instead of Hessian by changing the scheme from "hessian:" to "cxf:".

```
<web-app xmlns="http://caucho.com/ns/resin">

  <servlet-mapping url-pattern="/hello" servlet-class="qa.MyService">
    <protocol uri="cxf:"/>
  </servlet-mapping>

</web-app>
```

Example: WEB-INF/resin-web.xml - CXF Service

Client servlet - using the protocol

On the client side, the application needs a proxy to invoke the server methods. Since the protocols themselves are generic, the client will work with any server even if written in a different language like C#, as long as the protocol is compatible.

The client uses Resin's dependency injection to get a client proxy for the protocol. By using dependency injection, the client code remains independent of the protocol choice or any protocol-setup housekeeping. The only dependency is on the client API itself. Because the client uses the type-safe API, the Java compiler validates the protocol, making compatibility as certain as possible.

The `@javax.webbeans.In` annotation asks Resin to lookup the `qa.Hello` client stub that's been configured and gives it to the servlet. Since the client stub is thread-safe it can be safely used in the servlet code.

```
package qa;

import java.io.*;
import javax.servlet.*;
import javax.webbeans.*;

public class MyServlet extends GenericServlet {
  @In private Hello _hello;

  public void service(ServletRequest request,
                      ServletResponse response)
    throws IOException
  {
    out.println("hello: " + _hello.hello());
  }
}
```

Example: qa/MyServlet.java - Servlet Client

Although most clients will just have a single `Hello` proxy and can use the `@In` tag, more complicated applications can use the `@Named` tag or even custom `@BindingTag` annotations to select the right proxy. The Resin IoC documentation has more details.

Configuration - selecting the protocol

To configure the client, you need to specify the protocol type, the URL, and the API class in a <`remote-client`> tag in the resin-web.xml. Since the client code uses injection to get the proxy, we can switch protocols if necessary.

The example uses Hessian, so the `uri` attribute uses a "hessian:" scheme with Hessian's `url` parameter. The API class is `qa.Hello` .

```
<web-app xmlns="http://caucho.com/ns/resin">

  <remote-client class="qa.Hello"
                 uri="hessian:url=http://localhost:8080/hello"/>

  <servlet-mapping url-pattern="/demo"
                   servlet-class="qa.MyServlet"/>

</web-app>
```

Example: WEB-INF/resin-web.xml - Hessian Client

Since the client code only depends on the proxy API, changing to use CXF (SOAP) just requires changing the protocol scheme from "hessian:" to "cxf:".

```
<web-app xmlns="http://caucho.com/ns/resin">

  <remote-client class="qa.Hello"
                 uri="cxf:url=http://localhost:8080/hello"/>

  <servlet-mapping url-pattern="/demo"
                   servlet-class="qa.MyServlet"/>

</web-app>
```

Example: WEB-INF/resin-web.xml - CXF Client

Available Protocols

Resin 3.1.5 has the following protocol drivers available:

- hessian: The Hessian protocol is a fast, compact binary protocol with implementations available in a large number of languages including Flash, PHP and C#.
- burlap: The Burlap protocol is the XML-based cousin of Hessian.
- cxf: The CXF driver uses the Apache CXF project for SOAP client and server protocols.

Protocol Plugin Architecture

You can extend Resin's remoting protocols by adding a plugin for either the server or client. In either case, the required API is deliberately kept simple.

Server plugins

```
public interface ProtocolServletFactory {

  public Servlet createServlet(Class serviceClass, Object service)
    throws ServiceException;

}
```

```
package com.caucho.remote.hessian;

import com.caucho.hessian.server.*;
import com.caucho.remote.*;
import com.caucho.remote.server.*;

import javax.servlet.*;

public class HessianProtocolServletFactory
  extends AbstractProtocolServletFactory
{
  public Servlet createServlet(Class serviceClass, Object service)
    throws ServiceException
  {
    HessianServlet servlet = new HessianServlet();

    servlet.setHome(service);
    servlet.setHomeAPI(getRemoteAPI(serviceClass));

    return servlet;
  }
}
```

Example: HessianProtocolServletFactory

Resin's URI aliases are configured by property files in `WEB-INF/services/com.caucho.config.uri` . Each interface has its property file specifying the implementation class for each URI scheme. In this case, the interface is `com.caucho.remote.server.ProtocolServletFactory` , so it the scheme mappings are added to a file with the same name:

```
burlap=com.caucho.remote.burlap.BurlapProtocolServletFactory
hessian=com.caucho.remote.hessian.HessianProtocolServletFactory
```

com.caucho.remote.server.ProtocolServletFactory

Client plugins

```
public interface ProtocolProxyFactory
{
  public Object createProxy(Class api);
}
```

```
package com.caucho.remote.hessian;

import com.caucho.hessian.client.*;
import com.caucho.remote.*;
import com.caucho.remote.client.*;

public class HessianProtocolProxyFactory
  extends AbstractProtocolProxyFactory
{
  private HessianProxyFactory _factory = new HessianProxyFactory();

  private String _url;

  public void setURL(String url)
  {
    _url = url;
  }

  public Object createProxy(Class api)
  {
    try {
      return _factory.create(api, _url);
    } catch (Exception e) {
      throw ServiceException(e);
    }
  }
}
```

Example: HessianProtocolProxyFactory.java

21.2 Hessian

Hessian Client

Using a Hessian service from a Java client is like calling a method. The HessianProxyFactory creates proxies which act like normal Java objects, with possibility that the method might throw a protocol exception if the remote connection fails. Using HessianProxyFactory requires JDK1.3.

Each service will have a normal Java interface describing the service. The trivial hello, world example just returns a string. Because the Hessian services support Java serialization, any Java type can be used.

```
package example;

public interface Basic {
  public String hello();
}
```

API for Basic service

The following is an example of a standalone Hessian client. The client creates a HessianProxyFactory. The client uses the factory to create client stubs with the given target URL and a Java interface for the API. The returned object is a stub implementing the API.

```
package example;

import com.caucho.hessian.client.HessianProxyFactory;

public class BasicClient {
  public static void main(String []args)
    throws Exception
  {
    String url = "http://www.caucho.com/hessian/test/basic";

    HessianProxyFactory factory = new HessianProxyFactory();
    Basic basic = (Basic) factory.create(Basic.class, url);

    System.out.println("Hello: " + basic.hello());
  }
}
```

Hessian Client for Basic service

That's it! There are no more complications to using the client. The service can add methods and use any Java type for parameters and results.

Hessian Service

While most Hessian services will use Resin-CMP or Resin-EJB, to take advantage of the benefits of EJB, the Hessian library makes it possible to write services by extending HessianServlet.

Any public method is treated as a service method. So adding new methods is as easy as writing a normal Java class.

Because the service is implemented as a Servlet, it can use all the familiar servlet data in the ServletContext, just like a normal servlet.

```
package example;

public class BasicService implements Basic {
  private String _greeting = "Hello, world";

  public void setGreeting(String greeting)
  {
    _greeting = greeting;
  }

  public String hello()
  {
    return _greeting;
  }
}
```

Hello Service

Configuration with Dependency Injection in Resin 3.0

```
<web-app xmlns="http://caucho.com/ns/resin">
  <servlet servlet-name="hello"
           servlet-class="com.caucho.hessian.server.HessianServlet">
    <init>
      <home resin:type="example.BasicService">
        <greeting>Hello, world</greeting>
      </home>

      <home-api>example.Basic</home-api>
    </init>
  </servlet>

  <servlet-mapping url-pattern="/hello"
                   servlet-name="hello"/>
</web-app>
```

resin-web.xml

Configuration for standard web.xml

Since the HessianServlet is a standard servlet, it can also be configured in the standard servlet configuration.

```
<web-app>
  <servlet>
   <servlet-name>hello</servlet-name>
   <servlet-class>com.caucho.hessian.server.HessianServlet</servlet-class>
    <init-param>
      <param-name>home-class</param-name>
      <param-value>example.BasicService</param-value>
    </init-param>
    <init-param>
      <param-name>home-api</param-name>
      <param-value>example.Basic</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <url-pattern>/hello</url-pattern>
    <servlet-name>hello</servlet-name>
  </servlet-mapping>
</web-app>
```

web.xml

Hessian Serialization

The Hessian classes can be used for serialization and deserialization. Hessian's serialization forms the basis for the protocol and taking control of the serialization lets application use Hessian more efficiently than the proxy interface for specialized application protocols.

```
Object obj = ...;

OutputStream os = new FileOutputStream("test.xml");
Hessian2Output out = new Hessian2Output(os);

out.writeObject(obj);
os.close();
```

Serialization

```
InputStream is = new FileInputStream("test.xml");
Hessian2Input in = new Hessian2Input(is);

Object obj = in.readObject(null);
is.close();
```

Deserialization

When serializing Java objects more complex than primitives or Strings, make sure the classes for those objects implement java.io.Serializable.

Hessian with large binary data

When a distributed application needs to send large amounts of binary data, it can be more efficient to use `InputStream` to avoid allocating large byte arrays. Only the final argument of the method may be an `InputStream` , since the data is read during invocation. For example, a file downloading service could be implemented efficiently using Hessian.

In this example, the client needs to take control of the Hessian protocol directly, because the proxy interface would require buffering the entire file before the call returns.

```
package example;

public interface Upload {
  public void upload(String filename, InputStream data);
}
```

file upload API

If the result is an InputStream, it is very important that the `InputStream.close()` be put in an `finally` block, because Hessian will not close the underlying HTTP stream until all the data is read and the input stream is closed.

```
package example;

public interface Download {
  public InputStream download(String filename, InputStream data);
}
```

file download API

```
InputStream is = fileProxy.download("test.xml");

try {
  ... // read data here
} finally {
  is.close();
}
```

Download Java Code

Hessian Client for a cell-phone

Hessian can be used for even small Java devices. The following classes from com.caucho.hessian.client can be extracted into a J2ME jar:

- MicroHessianInput

- MicroHessianOutput
- HessianRemote
- HessianServiceException
- HessianProtocolException

The following example shows the code for using a cell phone as a client. It's a bit more complicated than using the proxy, since the client is responsible for creating the connection and writing the data.

```
import javax.microedition.io.Connector;
import javax.microedition.io.HttpConnection;

...

MicroHessianInput in = new MicroHessianInput();

String url = "http://www.caucho.com/hessian/test/basic";

HttpConnection c = (HttpConnection) Connector.open(url);

c.setRequestMethod(HttpConnection.POST);

OutputStream os = c.openOutputStream();
MicroHessianOutput out = new MicroHessianOutput(os);

out.call("hello", null);

os.flush();

is = c.openInputStream();

MicroHessianInput in = new MicroHessianInput(is);
Object value = in.readReply(null);
```

```
Hello, world
```

Chapter 22

Messaging

22.1 Resin Messaging

Messaging hello, world

Resin's messaging is build around JMS, the Java messaging service API and EJB message driven beans. A simple messaging application can use the `BlockingQueue` API to send messages and implement a `MessageListener` to receive messages.

Because messaging is integrated with the Resin IoC container, applications can use standard WebBeans injection to obtain the queues, avoiding code dependencies and improving testing.

The following example sends a "hello, world" message from `MySendingServlet` and processes it in `MyListener` . The servlet does not wait for the listener, it completes immediately. The `MyListener` message bean will receive the message when it's available.

The `@Named` WebBeans annotation tells Resin to look for a configured `BlockingQueue` named "myQueue" and inject it into the `_queue` variable when the servlet is initialized. The `offer` method sends the message to the JMS queue using a JMS `ObjectMessage` .

```
package demo;

import java.io.*;
import javax.servlet.*;
import java.util.BlockingQueue;
import javax.webbeans.Named;

public MySendingServlet extends GenericServlet
{
  @Named("myQueue") private BlockingQueue _queue;

  public void service(ServletRequest req, ServletResponse res)
    throws IOException, ServletException
  {
    String msg = "hello, world";

    _queue.offer(msg);

    System.out.println("Sent: " + msg);
  }
}
```

demo/MySendingServlet.java

The EJB message bean service will receive the message and pass it along to `MyListener` . Since Resin's `BlockingQueue` API automatically wraps the object in a JMS `ObjectMessage` , `MyListener` needs to unwrap it.

```
package demo;

import javax.jms.*;

public MyListener implements MessageListener
{
  public void onMessage(Message message)
  {
    ObjectMessage oMsg = (ObjectMessage) message;

    System.out.println("Received: " + oMsg.getObject());
  }
}
```

demo/MyListener.java

Now that the code's written, we just need to configure it in the `WEB-INF/resin-web.xml` . The <jms-connection-factory> configures Resin as the JMS provider, the <jms-queue> configures a memory-based queue as the implementation, and the <ejb-message-bean> configures our listener.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-connection-factory uri="resin:"/>
  <jms-queue name="myQueue" uri="memory:"/>

  <ejb-message-bean class="demo.MyListener">
    <destination>#{myQueue}</destination>
  </ejb-message-bean>

  <servlet-mapping url-pattern="/test"
                   servlet-class="demo.MySendingServlet"/>

</web-app>
```

WEB-INF/resin-web.xml

JMS Queues

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-queue name="my-queue" uri="memory:"/>
  <jms-connection-factory uri="resin:"/>

</web-app>
```

resin-web.xml - queue configuration

Memory Queue

Resin's memory queue is a basic, non-persistent queue suitable for testing and for cases where losing the queue contents at a server crash is acceptable. Like Resin's other queues, you can use the `BlockingQueue` API to send messages, and use a simple listener to receive messages.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-connection-factory uri="resin:"/>
  <jms-queue name="myQueue" uri="memory:"/>

  <ejb-message-bean class="demo.MyListener">
    <destination>#{myQueue}</destination>
  </ejb-message-bean>

</web-app>
```

resin-web.xml - Memory queue and listener

File Queue

The file queue backs messages on the local filesystem, allowing for recovery in case of system crash. The saved file is efficient, using the same backing store as Resin's proxy caching and persistent sessions.

The file queue configuration requires an additional 'path' parameter to specify a directory for the backing files.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-queue name="myQueue" uri="file:path=WEB-INF/messaging"/>

  <jms-connection-factory uri="resin:"/>

  <ejb-message-bean class="demo.MyListener">
    <destination>#{myQueue}</destination>
  </ejb-message-bean>

</web-app>
```

resin-web.xml - file queue and listener

Cluster Server Queue

The cluster server queue is a file queue which can also receive messages from the local cluster. On the local machine, it acts exactly like the file queue. When used with the cluster client queue, clients can distribute messages to any server queue in the cluster, allowing for load balancing.

Like the file queue, the cluster server queue requires a 'path' attribute to specify the location of the backing file.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-queue name="myQueue" uri="server:">
    <init>
      <name>my-queue</name>
      <path>WEB-INF/jms</path>
    </init>
  </jms-queue>

  <jms-connection-factory uri="resin:"/>

  <ejb-message-bean class="demo.MyListener">
    <destination>#{myQueue}</destination>
  </ejb-message-bean>

</web-app>
```

resin-web.xml - cluster server queue and listener

Cluster Client Queue

The client queue distributes messages to server queues in a cluster. Normally, only the sending methods are used for the client queue; the receiving message beans are handled by the server queues.

The client queue needs to configure the cluster of the server queues. The cluster can be different from the client's own cluster.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-queue name="myQueue" uri="client:cluster=message-tier"/>

  <jms-connection-factory uri="resin:"/>

</web-app>
```

resin-web.xml - client queue

JMS Topics

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-topic name="my-topic" uri="memory:"/>
  <jms-connection-factory uri="resin:"/>

</web-app>
```

resin-web.xml - topic configuration

Memory Topic

Resin's memory topic is a basic, non-persistent topic suitable for testing and for cases where losing the topic contents at a server crash is acceptable. Like Resin's other topics, you can use the `BlockingQueue` API to send messages, and use a simple listener to receive messages.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-connection-factory uri="resin:"/>
  <jms-topic name="myTopic" uri="memory:"/>

  <ejb-message-bean class="demo.MyListener">
    <destination>#{myTopic}</destination>
  </ejb-message-bean>

</web-app>
```

resin-web.xml - Memory topic and listener

File Topic

The file topic backs messages on the local filesystem for persistent subscriptions. Non-persistent subscriptions use the memory topic interface. The saved file is efficient, using the same backing store as Resin's proxy caching and persistent sessions.

The file topic configuration requires an additional 'path' parameter to specify a directory for the backing files.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-topic name="myTopic" uri="file:path=WEB-INF/messaging"/>

  <jms-connection-factory uri="resin:"/>

  <ejb-message-bean class="demo.MyListener">
    <destination>#{myTopic}</destination>
  </ejb-message-bean>

</web-app>
```

resin-web.xml - file topic and listener

ConnectionFactory

The `ConnectionFactory` resource defines the JMS factory for creating JMS connections.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-connection-factory name="jms/factory" uri="resin:"/>

</web-app>
```

resin-web.xml - ConnectionFactory resource

BlockingQueue API

Resin's queues implement the `java.util.concurrent.BlockingQueue` API. Since the queues are registered with Resin-IoC it's possible to use the `BlockingQueue` API directly without the JMS API.

```
package example;

import java.util.concurrent.BlockingQueue;
import java.io.*;
import javax.servlet.*;
import javax.webbeans.*;

public class TestServlet extends GenericServlet {
  private @In BlockingQueue _queue;

  public void service(ServletRequest req, ServletResponse res)
    throws IOException, ServletException
  {
    PrintWriter out = res.getWriter();

    _queue.offer("test message");

    out.println("receive: " + _queue.poll());
  }
}
```

TestServlet for JMS/BlockingQueue

The resin-web.xml configuration for the `BlockingQueue` API is simple, only requiring the <jms-queue> tag. Because the `BlockingQueue` uses Resin's JMS queue implementation directly, it already knows where to get the `ConnectionFactory` .

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-queue uri="memory:"/>

</web-app>
```

resin-web.xml configuration for memory BlockingQueue

BlockingQueue for 3rd party JMS

The `BlockingQueue` API is also available for other JMS providers. You'll need to configure a `JmsBlockingQueue` in the resin-web.xml to take advantage of it.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <resource-adapter class="org.apache.activemq.ra.ActiveMQResourceAdapter">
      <init server-url="vm://localhost"/>
  </resource-adapter>

   <connection-factory uri="activemq:" name="factory"/>

   <jms-queue uri="activemq:" name="queue">
       <init physicalName="queue.test"/>
   </jms-queue>

  <bean name="test" class="com.caucho.jms.queue.JmsBlockingQueue">
    <init>
      <factory>${factory}</factory>
      <destination>${queue}</destination>
    </init>
  </bean>

</web-app>
```

Example: BlockingQueue WEB-INF/resin-web.xml

Message Driven Beans

At some point, the application needs to receive messages from the and process them. Message driven beans provides a reliable, pooled framework for receiving messages. Applications just need to implement a simple listener interface, and register to listen with a queue or topic.

```
package javax.jms;

public interface MessageListener {

  public void onMessage(Message message);

}
```

The implementation class can use any Resin IoC capability, including injection, transaction annotations or interception. For example, a simple listener might use Amber/JPA to store messages in a database.

```
package demo;

import javax.jms.*;
import javax.persistence.*;
import javax.webbeans.*;

public class MyListener implements MessageListener
{
  private @In EntityManagerFactory _factory;

  public void onMessage(Message msg)
  {
    ObjectMessage oMsg = (ObjectMessage) msg;

    String value = oMsg.getObject();

    EntityManager em = _factory.createEntityManager();

    try {
      em.persist(new MyEntry(value));
    } finally {
      em.close();
    }
  }
}
```

demo/MyListener.java

The configuration in the resin-web.xml file will connect the `MyListener` class with the queue. In this case, we'll use the simple memory queue.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-connection-factory uri="resin:"/>
  <jms-queue name="my_queue" uri="memory:"/>

  <ejb-message-bean class="demo.MyListener">

    <destination>${my_queue}</destination>

  </ejb-message-bean>

</web-app>
```

WEB-INF/resin-web.xml

The <`ejb-message-bean`> tag configures the message bean pool with the application's listener specified by the `class` . Resin will automatically create several `MyListener` instances to process any queue messages. The <`destination`> tag specifies the queue to use.

Because the <`ejb-message-bean`> is a Resin IoC bean, it can use an optional <`init`> block to configure any parameters of `MyListener` .

JCA - Java Connector Architecture (.rar files)

The Java Connector Architecture is a driver architecture which connects JMS providers like ActiveMQ with Resin's message driven beans and JMS sessions. The JCA driver will configure a resource adapter and an activation spec to select a queue.

The resource adapter is the JCA driver's main service. It handles threading, socket connections, and creates any endpoints.

JCA for message driven beans

The activation specification configures the JCA driver with a message-driven bean. The configuration looks like:

```
<web-app xmlns="http://caucho.com/ns/resin">

  <resource-adapter class="org.apache.activemq.ra.ActiveMQResourceAdapter">
    <init server-url="vm://localhost"/>
  </resource>

  <ejb-message-bean class="qa.MyListener">

    <activation-spec class="org.apache.activemq.ra.ActiveMQActivationSpec">
      <init physical-name="queue.test"/>
    </activation-spec>

  </ejb-message-bean>

</web-app>
```

WEB-INF/resin-web.xml ActiveMQ

Resin can also provide shortcuts for the driver classes using the uri syntax:

```
<web-app xmlns="http://caucho.com/ns/resin">

  <resource-adapter uri="activemq:">
    <init server-url="vm://localhost"/>
  </resource>

  <ejb-message-bean class="qa.MyListener">

    <activation-spec uri="activemq:">
      <init physical-name="queue.test"/>
    </activation-spec>

  </ejb-message-bean>

</web-app>
```

WEB-INF/resin-web.xml ActiveMQ

Third-party JMS providers

- ActiveMQ and Resin

Chapter 23

JSF - Java Server Faces

23.1 JSF - Java Server Faces

JSF State Management with Hessian

A JSF page may produce state that depending on the functionality of the page can be quite large as in the number of components and in the sheer volume of data associated with them. When serialized for sending to the client or replicating across servers in the cluster smaller state will produce numerous benefits ranging from more efficient use of bandwidth to lower memory requirements and lower CPU usage.

To achieve smaller size Resin's JSF State Management uses Hessian Serialization for packaging objects composing JSF state into a network-transferable object. As opposed to Java Serialization Hessian takes advantage of a more compact storage scheme specified for Hessian Protocol.

Fast JSF

While adhering to the JSF Spec Resin takes advantage of its custom built JSP code generator that is capable of recognizing JSF tags and, when UIComponent can be inferred, creating an instance of the component instead of creating an instance of the tag, and plugging the component instance directly into the Component Tree.

This is what the JSP code generated in Fast JSF mode might look like:

```
HtmlOutputText _jsp_comp_1
  = Utils.add(jsf_context, request, response, HtmlOutputTest.class);
_jsp_comp_1.setValueExpression("value", _value_expr_0);
```

Fast-JSF JSP Generated Code

as opposed to the code generated in regular mode:

```
_jsp_HtmlOutputTextTag_1 = new HtmlOutputTextTag();
_jsp_HtmlOutputTextTag_1.setPageContext(pageContext);
_jsp_HtmlOutputTextTag_1.setParent((javax.servlet.jsp.tagext.Tag) _jsp_FacesViewTag_0);
_jsp_HtmlOutputTextTag_1.setJspId("jsp2");
_jsp_HtmlOutputTextTag_1.setValue(pageContext.createExpr(_value_expr_0,
                                    "#{msgs.title}", java.lang.Object.class));
_jsp_HtmlOutputTextTag_1.doStartTag();
_jsp_HtmlOutputTextTag_1.doEndTag();
```

Regular JSP Generated Code

As you can see from the above Resin will produce code that creates less objects, easing the load on JVM Garbage Collection, and making fewer calls, easing the load on CPU during page creation.

Setting Resin's Fast JSF

Setting mode to Fast JSF is done via using a fast-jsf flag in WEB-INF/resin-web.xml as in the code below:

```
<web-app xmlns="http://caucho.com/ns/resin">
  <jsp fast-jsf='true'/>
</web-app>
```

Fast-JSF resin-web.xml

Resin will automatically infer correct UIComponent type for all Actions supplied with Resin which includes all JSF 1.2 standard Actions from http://java.sun.com/jsf/html and http://java.sun.com/core spaces. Enabling custom components to take advantage of fast-jsf takes creating a mapping file that will be located by Resin at application startup. The file might look like the following:

```
<jsf-taglib xmlns="http://caucho.com/ns/resin">
  <uri>http://java.sun.com/jsf/html</uri>

  <jsf-tag>
    <name>column</name>
    <component-class>javax.faces.component.html.HtmlColumn</component-class>
  </jsf-tag>
  ...
  <jsf-tag>
    <name>commandButton</name>
    <component-class>javax.faces.component.html.HtmlCommandButton</component-class>
  </jsf-tag>
<jsf-taglib>
```

ftld for custom Actions

The mapping file needs to have `ftld` extension. Resin will look for files with ftld extension on web application's classpath. E.g. `org/ajax4jsf/taglib/ajax4jsf.ftld`

The children of the `jsf-taglib` tag do the following

- Element `uri` maps to the taglib uri from the corresponding `.tld` file
- Element `jsf-tag` defines a mapping by binding a tag `name` to `component-class`

If you run into a case where supplying an `.ftld` and setting `fast-jsf` to true does not work for your library please provide feedback asking in the forums or mailing list. A fallback to to non fast-jsf mode is always available via setting the value of fast-jsf to false.

JSF with Web Beans (JSR 299)

Web Beans can significantly reduce XML configuration required for your web application. By using annotations from `javax.webbeans` package a Java Bean can be turned into a Web Bean by supplying it with a @Component annotation like so:

```
package example;

import javax.webbeans.*;
@Component
public class FooBean {
}
```

Use @Component to make Java Bean a WebBean

Annotations `@Named` and one of `@ApplicationScoped` , `@ConversionScoped` , `@SessionScoped` , `@RequestScoped` , `@Dependant` (parent's scope) will give the Web Bean a name and scope:

```
package example;

import javax.webbeans.*;

@Component
@Named("foo")
@SessionScoped

public class FooBean {
}
```

Use @Named and @SessionScope to name a WebBean and specify its scope

The following block of annotation `@Component @Named("foo") @SessionScoped` atop of a bean is equivalent to the XML fragment below that would need to be placed into faces-config.xml

```
<managed-bean>
<managed-bean-name>quiz</managed-bean-name>
<managed-bean-class>com.corejsf.QuizBean</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Don't forget to place an emtpy web-beans.xml file into your classpath under `META-INF` directory to enable Web Beans for the context.

```
<web-beans xmlns="http://caucho.com/ns/resin"/>
```

e.g. WEB-INF/classes/META-INF/web-beans.xml

Web Beans has become a technology integrated into Resin's core and will work well with other Caucho's technologies. You can learn more about Web Beans and Resin at Resin IOC

JSF Developer Aid

Since version 3.2.0 resin offers JSF Developer Aid that allows to quickly introspect state of a Component Tree captured at the end of each phase and displayed in a tabbed view. The EL Experessions displayed in the view are navigable to screens that display backing data.

Alone with the state, the Aid displays information for all request headers and parameters, and, attributes set on request, session and application.

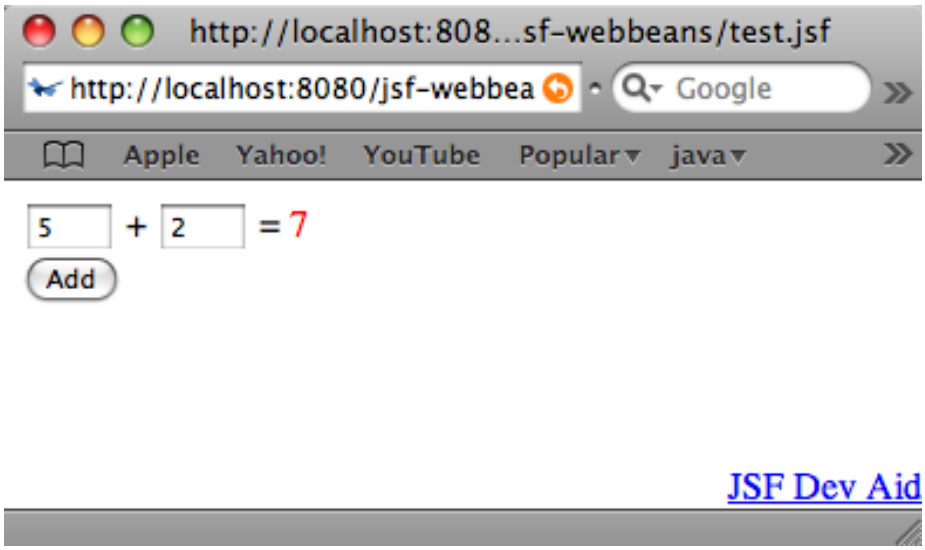
Enabling the JSF Developer Aid

To enable the Aid place a line <`jsf enable-developer-aid='true'/`> into `resin-web.xml` file.

```
<web-app xmlns='http://caucho.com/ns/resin'>
  <jsf enable-developer-aid='true'/>
</web-app>
```

WEB-INF/resin-web.xml

Once the Aid is enabled a *JSF Dev Aid* link will appear in the right bottom corner of a page as on the picture below.



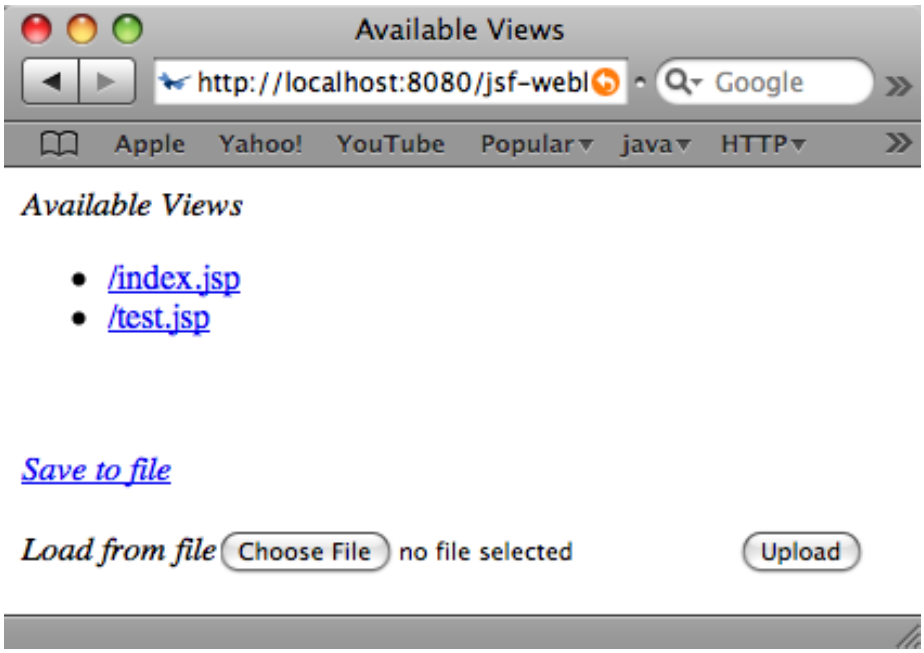
If desired, the position of the link can be changed via a `developer-aid-link-style` parameter in resin-web.xml file.

```
<web-app xmlns='http://caucho.com/ns/resin'>
  <jsf enable-developer-aid='true' developer-aid-link-style='position:absolute; bottom:1; right:1'/>
</web-app>
```

Link position

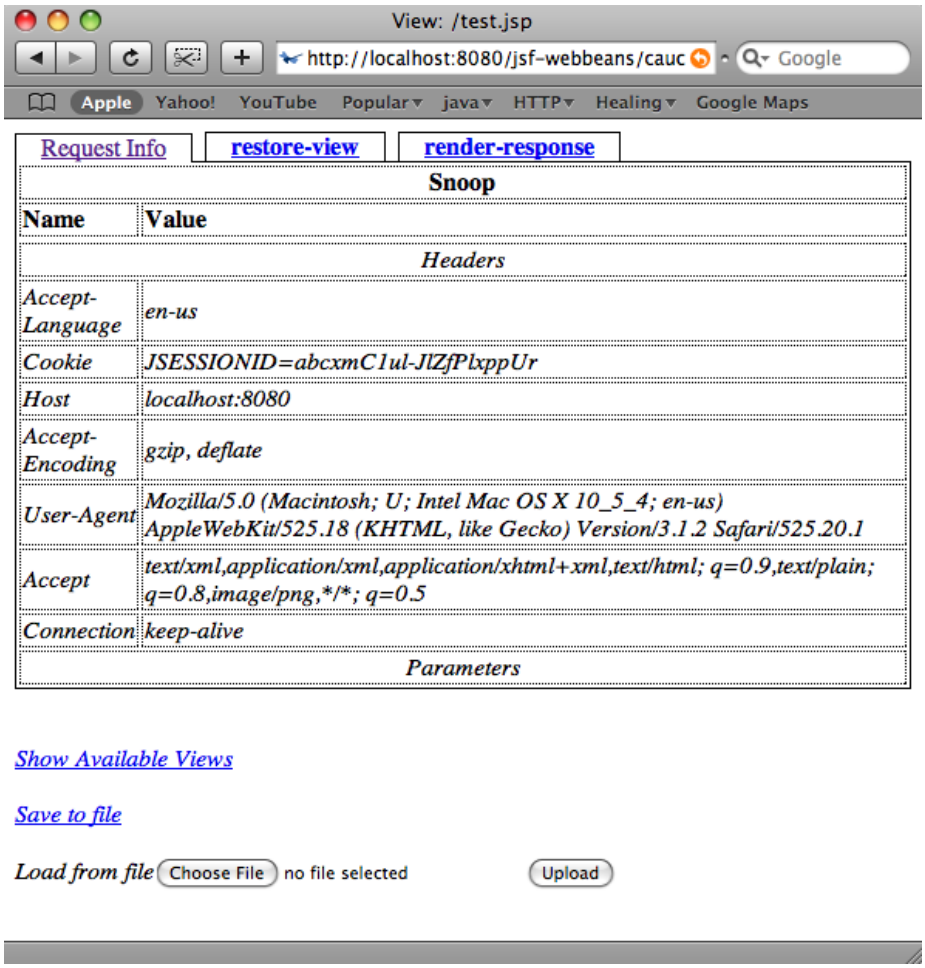
Aid's main page

The main page lists available views and controls that allow saving the information into a binary file or loading from a file saved earlier. Persisted to a file, captured state may be shared with other developers, attached to QA bug reports and so on.



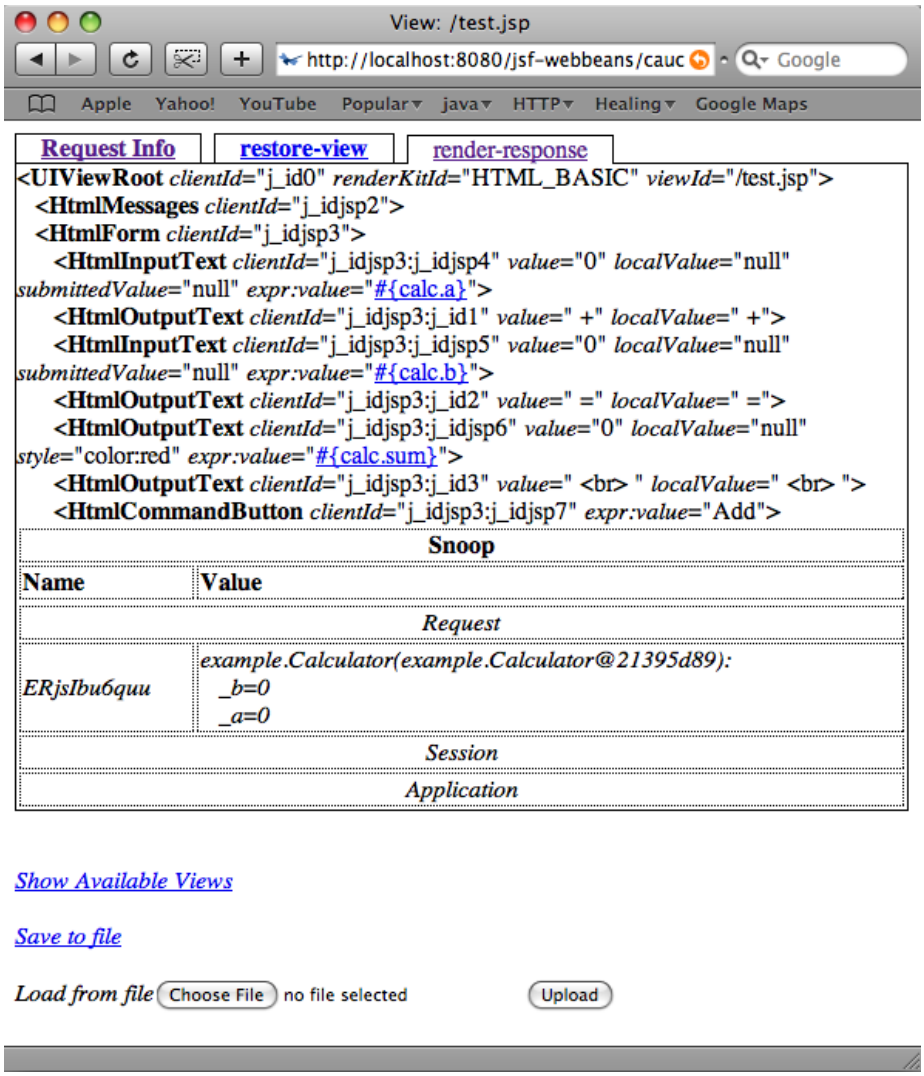
Aid's Request Info page

Request Info page will show headers and parameters contained in HTTP request.



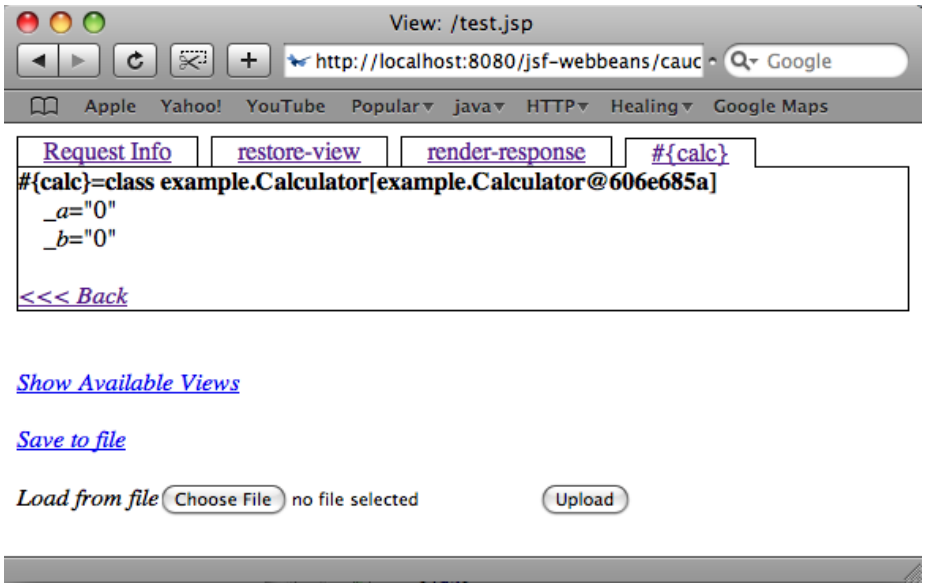
Aid's View States

The aid captures state of the `UIViewRoot` at the end of each phase and offers it in a tabbed view. Alone with it, the aid captures attributes set on request, session and application.



Aid's EL Expression Display

Expressions encoutered during introspection of a Component Tree become links and navigate to a page that display result of evaluation of that expression.


View: /test.jsp
http://localhost:8080/jsf-webbeans/cauc
Google
Apple
Yahoo!
YouTube
Popular
java
HTTP
Healing
Google Maps
Request Info
restore-view
render-response
#{calc}
#{calc}=class example.Calculator[example.Calculator@606e685a]
_a="0"
_b="0"
<<< Back
Show Available Views
Save to file
Load from file
Choose File
no file selected
Upload

Chapter 24

Configuration Tags

24.1 cluster: Cluster tag configuration

See Also

- See the index for a list of all the tags.
- See Web Application configuration for web.xml (Servlet) configuration.
- See Server tags for ports, threads, and JVM configuration.
- See Resource configuration for resources: classloader, databases, connectors, and resources.
- See Log configuration for access log configuration, java.util.logging, and stdout/stderr logging.

24.1.1 <access-log>

<access-log> configures a HTTP access log for all virtual hosts in the cluster. See access-log in the <host> tag for more information.

24.1.2 <cache>

child of: cluster

<cache> configures the proxy cache (requires Resin Professional). The proxy cache improves performance by caching the output of servlets, jsp and php pages. For database-heavy pages, this caching can improve performance and reduce database load by several orders of magnitude.

The proxy cache uses a combination of a memory cache and a disk-based cache to save large amounts of data with little overhead.

Management of the proxy cache uses the ProxyCacheMXBean.

ATTRIBUTE	DESCRIPTION	DEFAULT
path	Path to the persistent cache files.	cache/
disk-size	Maximum size of the cache saved on disk.	1024M
enable	Enables the proxy cache.	true
enable-range	Enables support for the HTTP Range header.	true
entries	Maximum number of pages stored in the cache.	8192
max-entry-size	Largest page size allowed in the cache.	1M
memory-size	Maximum heap memory used to cache blocks.	8M
rewrite-vary-as-private	Rewrite Vary headers as Cache-Control: private to avoid browser and proxy-cache bugs (particularly IE).	false

<cache> Attributes

```
element cache {
  disk-size?
  & enable?
  & enable-range?
  & entries?
  & path?
  & max-entry-size?
  & memory-size?
  & rewrite-vary-as-private?
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="web-tier">
        <cache entries="16384" disk-size="2G" memory-size="256M"/>

        <server id="a" address="192.168.0.10"/>

        <host host-name="www.foo.com">
    </cluster>
</resin>
```

Example: enabling proxy cache

24.1.3 <cluster>

child of: resin

<cluster> configures a set of identically-configured servers. The cluster typically configures a set of <server>s, each with some ports, and a set of virtual <host>s.

Only one <cluster> is active in any on server. At runtime, the <cluster> is selected by the <server> with `id` matching the -server-id on the command line.

ATTRIBUTE	DESCRIPTION	DEFAULT
id	The cluster identifier.	required
access-log	An access-log shared for all virtual hosts.	
cache	Proxy cache for HTTP-cacheable results.	
connection-error-page	IIS error page to use when isapi_srun to Resin connection fails	
ear-default	default values for deployed ear files	
error-page	Custom error-page when virtual-hosts fail to match	
host	Configures a virtual host	
host-default	Configures defaults to apply to all virtual hosts	
host-deploy	Automatic host deployment based on a deployment directory	
ignore-client-disconnect	Ignores socket exceptions thrown because browser clients have prematurely disconnected	false
invocation-cache-size	Size of the system-wide URL to servlet invocation mapping cache	16384
invocation-cache-max-url-length	Maximum URL length saved in the invocation cache	256
machine	Configuration for grouping <server> onto physical machines	

persistent-store	Configures the distributed/persistent store	
ping	Periodic checking of server URLs to verify server activity	
redeploy-mode	"automatic" or "manual"	automatic
resin:choose	Conditional configuration based on EL expressions	
resin:import	Imports a custom cluster.xml files for a configuration management	
resin:if	Conditional configuration based on EL expressions	
rewrite-dispatch	rewrites and dispatches URLs using regular expressions, similar to mod_rewrite	
root-directory	The root filesystem directory for the cluster	${resin.root}
server	Configures JVM instances (servers). Each cluster needs at least one server	
server-default	Configures defaults for all server instances	
server-header	Configures the HTTP "Server: Resin/xxx" header	Resin/Version
session-cookie	Configures the servlet cookie name	JSESSIONID
session-sticky-disable	Disables sticky-sessions on the load balancer	false
url-character-encoding	Configures the character encoding for URLs	utf-8
url-length-max	Configures the maximum length of an allowed URL	8192
web-app-default	Configures defaults to apply to all web-apps in the cluster	

<cluster> Attributes

```
element cluster {
  attribute id { string }
  & \hyperlink{env-tags.xtp}{environment resources}
  & access-log?
  & cache?
  & connection-error-page?
  & ear-default*
  & error-page*
  & host*
  & host-default*
  & host-deploy*
  & ignore-client-disconnect?
  & invocation-cache-size?
  & invocation-cache-max-url-length?
  & machine*
  & persistent-store?
  & ping*
  & redeploy-mode?
  & resin:choose*
  & resin:import*
  & resin:if*
  & rewrite-dispatch?
  & root-directory?
  & server*
  & server-default*
  & server-header?
  & session-cookie?
  & session-sticky-disable?
  & url-character-encoding?
  & url-length-max?
  & web-app-default*
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="web-tier">
        <server-default>
            <http port="8080"/>
        </server-default>

        <server id="a" address="192.168.0.10"/>
        <server id="b" address="192.168.0.11"/>

        <host host-name="www.foo.com">
          ...
        </host>
    </cluster>
</resin>
```

Example: cluster-default

rewrite-vary-as-private

Because not all browsers understand the Vary header, Resin can rewrite Vary to a Cache-Control: private. This rewriting will cache the page with the Vary in Resin's proxy cache, and also cache the page in the browser. Any other proxy caches, however, will not be able to cache the page.

The underlying issue is a limitation of browsers such as IE. When IE sees a Vary header it doesn't understand, it marks the page as uncacheable. Since IE only understands "Vary: User-Agent", this would mean IE would refuse to cache gzipped pages or "Vary: Cookie" pages.

With the <rewrite-vary-as-private> tag, IE will cache the page since it's rewritten as "Cache-Control: private" with no Vary at all. Resin will continue to cache the page as normal.

24.1.4 <cluster-default>

child of: resin

<cluster-default> defines default cluster configuration for all clusters in the <resin> server.

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster-default>
        <cache entries="16384" memory-size="64M"/>
    </cluster-default>

    <cluster id="web-tier">
        ...
    </cluster>

    <cluster id="app-tier">
        ...
    </cluster>
</resin>
```

Example: cluster-default

24.1.5 <connection-error-page>

child of: cluster

<connection-error-page> specifies an error page to be used by IIS when it can't contact an app-tier Resin. This directive only applies to IIS.

```
element connection-error-page {
  string
}
```

24.1.6 <development-mode-error-page>

child of: cluster

<development-mode-error-page> enables browser error reporting with extra information. Because it can expose internal data, it is not generally recommended in production systems. (The information is generally coped to the log.

24.1.7 <ear-default>

child of: cluster

<ear-default> configures defaults for .ear resource, i.e. enterprise applications.

24.1.8 <error-page>

child of: cluster

<error-page> defines a web page to be displayed when an error occurs outside of a virtual host or web-app. Note, this is not a default error-page, i.e. if an error occurs inside a <host> or <web-app>, the error-page for that host or web-app will be used instead.

See webapp: error-page.

```
element error-page {
  (error-code | exception-type)?
  & location?
}
```

24.1.9 <host>

child of: cluster

<host> configures a virtual host. Virtual hosts must be configured explicitly.

- See host tags for configuration details.

ATTRIBUTE	DESCRIPTION	DEFAULT
id	primary host name	none
regexp	Regular expression based host matching	none
host-name	Canonical host name	none
host-alias	Aliases matching the same host	none
secure-host-name	Host to use for a redirect to SSL	none
root-directory	Root directory for host files	parent directory

startup-mode	'automatic', 'lazy', or 'manual', see Startup and Redeploy Mode	automatic

<host> attributes

```
<host host-name="www.foo.com">
  <host-alias>foo.com</host-alias>
  <host-alias>web.foo.com</host-alias>

  <root-directory>/opt/www/www.foo.com</root-directory>

  <web-app id="/" document-directory="webapps/ROOT">

  </web-app>
  ...
</host>
```

Example: explicit host

```
<host regexp="([^.]+)\.foo\.com">
  <host-name>${host.regexp[1]}.foo.com</host-name>

  <root-directory>/var/www/hosts/www.${host.regexp[1]}.com</root-directory>

  ...
</host>
```

Example: regexp host

It is recommended that any <host> using a regexp include a <host-name> to set the canonical name for the host.

24.1.10 <host-default>

child of: cluster

Defaults for a virtual host.

The host-default can contain any of the host configuration tags. It will be used as defaults for any virtual host.

24.1.11 <host-deploy>

child of: cluster

Configures a deploy directory for virtual host.

The host-deploy will add an EL variable ${name}, referring to the name of the host jar file.

ATTRIBUTE	DESCRIPTION	DEFAULT
path	path to the deploy directory	required
expand-path	path to the expansion directory	path
host-default	defaults for the expanded host	
host-name	the host name to match	${name}

24.1.12 <ignore-client-disconnect>

child of: cluster**default:** true

ignore-client-disconnect configures whether Resin should ignore disconnection exceptions from the client, or if it should send those exceptions to the application.

```
element ignore-client-disconnect {
  r_boolean-Type
}
```

24.1.13 <invocation-cache-size>

child of: cluster**default:** 8192

Configures the number of entries in the invocation cache. The invocation cache is used to store pre-calculated servlet and filter chains from the URLs. It's also used as the basis for proxy caching.

```
element invocation-cache-size {
  r_int-Type
}
```

24.1.14 <invocation-cache-max-url-length>

child of: cluster**default:** 256

Configures the longest entry cacheable in the invocation cache. It is used to avoid certain types of denial-of-service attacks.

```
element invocation-cache-max-url-length {
  r_int-Type
}
```

24.1.15 <persistent-store>

child of: cluster

Defines the cluster-aware persistent store used for sharing distributed sessions. The allowed types are "jdbc", "cluster" and "file". The "file" type is only recommended in single-server configurations.

The <persistent-store> configuration is in the <server> level because it needs to share update information across the active cluster and the <cluster> definition is at the <server> level. Sessions activate the persistent store with the <use-persistent-store> tag of the <session-config>.

See Persistent sessions for more details.

ATTRIBUTE	DESCRIPTION	DEFAULT
init	initialization parameters for the persistent-store	
type	cluster, jdbc, or file	required

<persistent-store> Attributes

```
element persistent-store {
  type
  & init?
}
```

cluster store

The cluster store shares copies of the sessions on multiple servers. The original server is used as the primary, and is always more efficient than the backup servers. In general, the cluster store is preferred because it is more scalable, and with the "triplicate" attribute, the most reliable..

ATTRIBUTE	DESCRIPTION	DEFAULT
always-load	Always load the value	false
always-save	Always save the value	false
max-idle-time	How long idle objects are stored (session-timeout will invalidate items earlier)	24h
path	Directory to store the objects	required
save-backup	Saves backup copies of all distributed objects (3.2.0).	true

save-triplicate	Saves three copies of all distributed objects (3.2.0).	true
wait-for-acknowledge	Requires the sending server to wait for all acks.	false

cluster tags

```
element persistent-store {
  type { "cluster "}

  element init {
    always-load?
    & always-save?
    & max-idle-time?
    & triplicate?
    & wait-for-acknowledge?
  }
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster>
  <server id="a" address="192.168.0.1" port="6800"/>
  <server id="b" address="192.168.0.2" port="6800"/>

  <persistent-store type="cluster">
    <init>
      <triplicate>true</triplicate>
    </init>
  </persistent-store>

  <web-app-default>
    <session-config use-persistent-store="true"/>
  </web-app-default>
</cluster>
</resin>
```

Example: cluster store

jdbc store

The JDBC store saves sessions in a JDBC database. Often, this will be a dedicated database to avoid overloading the main database.

ATTRIBUTE	DESCRIPTION	DEFAULT
always-load	Always load the value	false
always-save	Always save the value	false

blob-type	Schema type to store values	from JDBC meta info
data-source	The JDBC data source	required
table-name	Database table	persistent_session
max-idle-time	How long idle objects are stored	24h

jdbc store Attributes

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster>
  <server id="a" address="192.168.0.1" port="6800"/>
  <server id="b" address="192.168.0.2" port="6800"/>

  <persistent-store type="jdbc">
    <init>
      <data-source>jdbc/session</data-source>

      <max-idle-time>24h</max-idle-time>
    </init>
  </persistent-store>

  <web-app-default>
    <session-config use-persistent-store="true"/>
  </web-app-default>
</cluster>
</resin>
```

Example: jdbc-store

file store

The file store is a persistent store for development and testing or for single servers. Since it is not aware of the clusters, it cannot implement true distributed objects.

ATTRIBUTE	DESCRIPTION	DEFAULT
always-load	Always load the value	false
always-save	Always save the value	false
max-idle-time	How long idle objects are stored	24h
path	Directory to store the sessions	required

file tags

24.1.16 <ping>

child of: cluster

Starts a thread that periodically makes a request to the server, and restarts Resin if it fails. This facility is used to increase server reliability - if there is a problem with the server (perhaps from a deadlock or an exhaustion of resources), the server is restarted.

A failure occurs if a request to the url returns an HTTP status that is not 200.

Since the local process is restarted, it does not make sense to specify a url that does not get serviced by the instance of Resin that has the ping configuration. Most configurations use url's that specify 'localhost' as the host.

This pinging only catches some problems because it's running in the same process as Resin itself. If the entire JDK freezes, this thread will freeze as well. Assuming the JDK doesn't freeze, the PingThread will catch errors like deadlocks.

ATTRIBUTE	DESCRIPTION	DEFAULT
url	A url to ping.	required
sleep-time	Time to wait between pings. The first ping is always 15m after the server starts, this is for subsequent pings.	15m
try-count	If a ping fails, number of times to retry before giving up and restarting	required
retry-time	time between retries	1s
socket-timeout	time to wait for server to start responding to the tcp connection before giving up	10s

<ping> Attributes

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">
    <cluster id="app-tier">
        <ping url="http://localhost/"/>
        ...
    </cluster>
</resin>
```

Example: resin.xml - simple usage of server ping

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">
    ...
    <cluster id="app-tier">
        <ping>
            <url>http://localhost:8080/index.jsp</url>
            <url>http://localhost:8080/webapp/index.jsp</url>
            <url>http://virtualhost/index.jsp</url>
            <url>http://localhost:443/index.jsp</url>

            <sleep-time>5m</sleep-time>
            <try-count>5</try-count>

            <!-- a very busy server -->
            <socket-timeout>30s</socket-timeout>
        </ping>
      ...
    </cluster>
</resin>
```

Example: resin.xml - configured usage of server ping

The class that corresponds to <ping> is PingThread.

Mail notification when ping fails

A refinement of the ping facility sends an email when the server is restarted.

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">
  ...
  <cluster id="web-tier">
    <ping resin:type="com.caucho.server.admin.PingMailer">
      <url>http://localhost:8080/index.jsp</url>
      <url>http://localhost:8080/webapp/index.jsp</url>

      <mail-to>fred@hogwarts.com</mail-to>
      <mail-from>resin@hogwarts.com</mail-from>
      <mail-subject>Resin ping has failed for server ${'${'}server.name}</mail-subject>
    </ping>
    ...
  </server>
</resin>
```

resin.xml - mail notification when ping fails

The default behaviour for sending mail is to contact a SMTP server at host 127.0.0.1 (the localhost) on port 25. System properties are used to configure a different SMTP server.

```
  <system-property mail.smtp.host="127.0.0.1"/>
  <system-property mail.smtp.port="25"/>
```

resin.xml - smtp server configuration

24.1.17 Resource Tags

child of: cluster

All Environment tags are available to the <cluster>. For example, resources like <database>.

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="app-tier">
        <database jndi-name="jdbc/test">
            <driver type="org.postgresql.Driver">
                <url>jdbc:postgresql://localhost/test</url>
                <user>caucho</user>
            </driver>
        </database>

        <server id="a" ...>
          ...

        <host host-name="www.foo.com">
          ...
    </cluster>
</resin>
```

Example: cluster environment

24.1.18 <rewrite-dispatch>

child of: cluster

<rewrite-dispatch> defines a set of rewriting rules for dispatching and forwarding URLs. Applications can use these rules to redirect old URLs to their new replacements.

See rewrite-dispatch for more details.

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="web-tier">

        <rewrite-dispatch>
            <redirect regexp="^http://www.foo.com"
                      target="http://bar.com/foo"/>
        </rewrite-dispatch>

    </cluster>
</resin>
```

rewrite-dispatch

24.1.19 <root-directory>

child of: cluster**default:** The root-directory of the <resin> tag.

<root-directory> configures the root directory for files within the cluster. All paths in the <cluster> will be relative to the root directory.

```
element root-directory {
  r_path-Type
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="app-tier">
        <root-directory>/var/www/app-tier</root-directory>

        <server id="a" ...>

        <host host-name="www.foo.com">
    </cluster>
</resin>
```

Example: cluster root-directory

24.1.20 <server>

child of: cluster

The <server> tag configures a JVM instance in the cluster. Each <server> is uniquely identified by its `id` attribute. The `id` will match the -server-id command line argument.

See the full server configuration for more details of the <server> tag and its children.

The current server is managed with a ServerMXBean. The ObjectName is `resin:type=Server` .

Peer servers are managed with ServerConnectorMXBean. The ObjectName is `resin:type=ServerConnector,name=server-id` .

ATTRIBUTE	DESCRIPTION	DEFAULT
address	IP address of the cluster port	127.0.0.1
bind-ports-after-start	If true, listen to the ports only after all initialization has completed, allowing load-balance failover.	true
cluster-port	Configures the cluster port in detail, allowing for customization of timeouts, etc.	
group-name	Used by the watchdog to switch setgid before starting the Resin JVM instance for security.	
http	Adds a HTTP port (see port tags)	
id	Unique server identifier	required
java-exe	The specific Java executable for the watchdog to launch the JVM	java
jvm-arg	Adds a JVM argument when the watchdog launches Resin.	
jvm-classpath	Adds a JVM classpath when the watchdog launches Resin.	
keepalive-connection-time-max	The total time a connection can be used for requests and keepalives	10min
keepalive-max	The maximum keepalives enabled at one time.	128
keepalive-select-enable	Enables epoll/select for keepalive requests to reduce threads (unix only)	true
keepalive-timeout	Timeout for a keepalive to wait for a new request	15s
load-balance-connect-timeout	How long the load-balancer should wait for a connection to this server	5s

load-balance-idle-time	How long the load balancer can keep an idle socket open to this server (see keepalive-timeout)	keepalive-time - 2s
load-balance-recover-time	How long the load balancer should treat this server as dead after a failure before retrying	15s
load-balance-socket-timeout	timeout for the load balancer reading/writing to this server	65s
load-balance-warmup-time	Warmup time for the load-balancer to throttle requests before sending the full load	60s
load-balance-weight	relative weight used by the load balancer to send traffic to this server	100
memory-free-min	minimum memory allowed for the JVM before Resin forces a restart	1M
ping	Configures a periodic ping of the server to force restarts when non-responsive	
port	Configures the cluster port (shortcut for <cluster-port>)	6800
protocol	Adds a custom socket protocol, e.g. for IIOP or SNMP.	
shutdown-wait-max	The maximum of time to wait for a graceful Resin shutdown before forcing a close	60s
socket-timeout	The read/write timeout for the socket	65s
thread-max	The maximum number of threads managed by Resin (JVM threads will be larger because of non-Resin threads)	4096

thread-executor-thread-max	Limits the threads allocated to application ScheduledExecutors from Resin	
thread-idle-max	Maximum number of idle threads in the thread pool	10
thread-idle-min	Minimum number of idle threads in the thread pool	5
user-name	The setuid user-name for the watchdog when launching Resin for Unix security.	
watchdog-jvm-arg	Additional JVM arguments when launching the watchdog manager	
watchdog-port	The port for the watchdog-manager to listen for start/stop/status requests	6700

<server> Attributes

```
element server {
  attribute id { string }
  & address?
  & bind-ports-after-start?
  & cluster-port*
  & group-name?
  & http*
  & java-exe?
  & jvm-arg?
  & jvm-classpath?
  & keepalive-connection-time-max?
  & keepalive-max?
  & keepalive-select-enable?
  & keepalive-timeout?
  & load-balance-connect-timeout?
  & load-balance-idle-time?
  & load-balance-recover-time?
  & load-balance-socket-timeout?
  & load-balance-warmup-time?
  & load-balance-weight?
  & memory-free-min?
  & ping?
  & port?
  & protocol?
  & shutdown-wait-max?
  & socket-timeout?
  & thread-max?
  & thread-executor-task-max?
  & thread-idle-max?
  & thread-idle-min?
  & user-name?
  & watchdog-jvm-arg*
  & watchdog-port?
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="web-tier">
        <server id="a" address="192.168.0.10" port="6800">
          <http port="8080"/>
        </server>

        <server id="b" address="192.168.0.11" port="6800">
          <http port="8080"/>
        </server>

        <server id="c" address="192.168.0.12" port="6800">
          <http port="8080"/>
        </server>

        <host id="">
          ...
    </cluster>
</resin>
```

Example: server

24.1.21 <server-default>

child of: cluster

Defines default values for all <server> instances. See <server> configuration for more details.

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="web-tier">
        <server-default>
            <server-port>6800</server-port>

            <http port="8080"/>
        </server-default>

        <server id="a" address="192.168.0.10"/>
        <server id="b" address="192.168.0.11"/>
        <server id="c" address="192.168.0.12"/>

        <host id="">
          ...
    </cluster>
</resin>
```

Example: server

24.1.22 <server-header>

child of: cluster**default:** Resin/3.1.x

Configures the HTTP Server: header which Resin sends back to any HTTP client.

```
element server-header {
  string
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="web-tier">
        <server-header>MyServer/1.0</server-header>
    </cluster>
</resin>
```

server-header

24.1.23 <session-cookie>

child of: cluster**default:** JSESSIONID

Configures the cookie used for servlet sessions.

```
element session-cookie {
  string
}
```

24.1.24 <session-sticky-disable>

child of: cluster**default:** false

Disables sticky sessions from the load balancer.

```
element session-sticky-disable {
  r_boolean-Type
}
```

24.1.25 <session-url-prefix>

child of: cluster**default:** ;jsessionid=

Configures the URL prefix used for session rewriting.**Note:** Session rewriting is discouraged as a potential security issue.

```
element session-cookie {
  string
}
```

24.1.26 <ssl-session-cookie>

child of: cluster**default:** value of session-cookie

Defines an alternative session cookie to be used for a SSL connection. Having two separate cookies increases security.

```
element ssl-session-cookie {
  string
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="web-tier">
        <ssl-session-cookie>SSLJSESSIONID</ssl-session-cookie>
        ...
    </cluster>
</resin>
```

Example: ssl-session-cookie

24.1.27 <url-character-encoding>

child of: cluster**default:** UTF-8

Defines the character encoding for decoding URLs.

The HTTP specification does not define the character-encoding for URLs, so the server must make assumptions about the encoding.

```
element url-character-encoding {
  string
}
```

24.1.28 <web-app-default>

child of: cluster

<web-app-default> defines default values for any web-app in the cluster.

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="app-tier">

        <web-app-default>
            <servlet servlet-name="resin-php"
                     servlet-class="com.caucho.quercus.servlet.QuercusServlet"/>

            <servlet-mapping url-pattern="*.php"
                             servlet-name="resin-php"/>
        </web-app-default>

        <host id="">
          ...
    </cluster>
</resin>
```

Example: web-app-default

24.2 database: Database tag configuration

See Also

- See Environment configuration for resources: classloader, databases, connectors, and resources.
- See Database Configuration for a detailed overview of databases and Resin.

24.2.1 <connection-wait-time>

child of: database

<connection-wait-time> configures the time a `getConnection` call should wait when then pool is full before trying to create an overflow connection.**default:** 10m

24.2.2 <close-dangling-connections>

child of: database

<close-dangling-connections> closes open connections at the end of a request and logs a warning and stack trace.**default:** true

24.2.3 <driver>

child of: database

<driver> configures a database driver for a connection pool. The individual driver information is available from the driver vendor or in the database driver page.

The content of the driver tag configures bean properties of the driver class, e.g. url, user, password.

```
element driver {
  type,
  *
}
```

24.2.4 <database>

child of: resin, cluster, host, web-app

<database> configures a database as a `javax.sql.DataSource` and stores it in jndi with the given `jndi-name` .

connection-wait-time	When the pool it full, how long to wait before opening a new connection anyway.	10m

```
<web-app xmlns="http://caucho.com/ns/resin">
  <database jndi-name="jdbc/test">
    <driver type="com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource">
      <url>jdbc:mysql://localhost:3306/test</url>
      <user></user>
      <password></password>
    </driver>
  </database>
</web-app>
```

Example: mysql database

```
element database {
  jndi-name
  & connection-Database?
  & driver+
  & connection-wait-time
  & max-active-time
  & max-close-statements
  & max-connections
  & max-create-connections
  & max-idle-time
  & max-overflow-connections
  & max-pool-time
  & password
  & ping
  & ping-table
  & ping-query
  & ping-interval
  & prepared-statement-cache-size
  & save-allocation-stack-trace
  & spy
  & transaction-timeout
  & user
  & xa
  & xa-forbid-same-rm
  & wrap-statements
}
```

24.2.5 <max-active-time>

child of: database

<max-active-time> configures the maximum time a connection can be active before Resin will automatically close it. Normally, the max-active-time should not be configured, since Resin will also automatically close a connection at the end of a request.

Sites should generally leave max-active-time at the default.**default:** 6h

24.2.6 <max-close-statements>

child of: database

<max-close-statements> configures how many open statements Resin should save to for the connection close. Since the JDBC `Connection.close()` call automatically closes any open statements, Resin's database pool needs to keep track of any open statements to close them in case the application has forgotten. The <max-close-statements> is primarily needed for older database drivers implementing the `java.sql.Driver` interface.**default:** 256

24.2.7 <max-connections>

child of: database

<max-connections> configures the maximum number of open connections allowed for Resin's database pool. Sites can use <max-connections> to throttle the number of database connections for an overloaded server. When `max-connections` is reached and an application calls `getConnection` , Resin will wait connection-wait-time or until a connection is freed before allocating a new connection.**default:** 128

24.2.8 <max-create-connections>

child of: database

<max-create-connections> configures the maximum number of simultaneous connection creations. Since connection creation is slow and database access can be spiky, Resin's pool limits the number of new connections to the database at any time. Once a connection has succeeded, a new connection can proceed.**default:** 5

24.2.9 <max-idle-time>

child of: database

<max-idle-time> configures the maximum time a connection can remain idle before Resin automatically closes it. Since idle databases tie up resources, Resin will slowly close idle connections that are no longer needed.

Higher values of <max-idle-time> will connections to remain in the idle pool for a longer time. Lower values will close idle connections more quickly.**default:** 30s

24.2.10 <max-pool-time>

child of: database

<max-pool-time> configures the maximum time the connection can remain open. A connection could theoretically remain open, switching between active and idle, for an indefinite time. The <max-pool-time> allows a site to limit to total time of that connection.

Most sites will leave <max-pool-time> at the default.**default:** 24h

24.2.11 <password>

child of: database

<password> configures the database connection password. Sites requiring additional security for their passwords can use the resin:type attribute to configure a password decoder.

24.2.12 <ping>

child of: database

<ping> enables connection validation. When <ping> is enabled, Resin will test the connection with <ping-query> or <ping-table> before returning a connection to the user. If the connection fails the test, Resin will close it and return a new connection.

For efficiency, Resin will only validate the connection if it has been idle for longer than <ping-interval>. **default:** false

24.2.13 <ping-table>

child of: database

<ping-table> configures the database table Resin should use to verify if a connection is still valid when returned from the pool.

24.2.14 <ping-query>

child of: database

<ping-query> specifies the query to use for validating if a database connection is still valid when returned from the idle pool.

24.2.15 <ping-interval>

child of: database

<ping-interval> configures when Resin should validate an idle connection. Connections which have been idle for less than <ping-interval> are assumed to be still valid without validation. Connections idle for longer than <ping-interval> are validated.

Sites can force a validation by setting <ping-interval> to 0.**default:** 1s

24.2.16 <prepared-statement-cache-size>

child of: database

<prepared-statement-cache-size> configures how many prepared statements Resin should cache for each connection. Caching prepared statement can improve performance for some database drivers by avoiding repeated parsing of the query SQL.**default:** 0

24.2.17 <save-allocation-stack-trace>

child of: database

<save-allocation-stack-trace> helps debugging application with a missing `Connection.close()` by saving the stack trace where the `Connection.getConnection()` was called. When Resin detects that the connection has failed to close, it can then print the allocation stack trace, which is more informative for tracking down errors.

24.2.18 <spy>

child of: database

The <spy> tag is a very useful logging tag for debugging database problems. If <spy> is enabled, all database queries will be logged at the "fine" level. Applications can use <spy> to debug unexpected database queries, or to improve query performance.**default:** false

```
0.6:setString(1,1)
0.6:executeQuery(select o.DATA from my_bean o where o.ID=?)
```

Example: spy output

24.2.19 <transaction-timeout>

child of: database

<transaction-timeout> configures the maximum time a transaction can be alive before a mandatory rollback.**default:** -1

24.3 <host>: Virtual Host configuration

See Also

- See the index for a list of all the tags.
- See Web Application configuration for web.xml (Servlet) configuration.
- See Resource configuration for resources: classloader, databases, JMS, EJB, and IoC beans.
- See Log configuration for access log configuration, java.util.logging, and stdout/stderr logging.

24.3.1 <access-log>

child of: cluster, host, web-app

<access-log> configures the access log file.

As a child of , overrides the definition in the that the web-app is deployed in. As a child of , overrides the definition in the that the host is in.

ATTRIBUTE	DESCRIPTION	DEFAULT
archive-format	the format for the archive filename when a rollover occurs, see Rollovers.	see below
format	Access log format.	see below

hostname-dns-lookup	log the dns name instead of the IP address (has a performance hit).	false
path	Output path for the log entries, see "Log Paths".	required
rollover-period	how often to rollover the log. Specify in days (15D), weeks (2W), months (1M), or hours (1h). See Rollovers.	none
rollover-size	maximum size of the file before a rollover occurs, in bytes (50000), kb (128kb), or megabytes (10mb). See Rollovers.	1mb
resin:type	a class extending for custom logging	com.caucho.server.log.AccessLog
init	Resin-IoC initialization for the custom class	n/a

<access-log> Attributes

```
element access-log {
  auto-flush?
  & archive-format?
  & auto-flush-time?
  & exclude?
  & format?
  & path?
  & rollover-count?
  & rollover-period?
  & rollover-size?
  & init?
}
```

The default archive format is `path` + ".%Y%m%d" or `path` + ".%Y%m%d.%H" if rollover-period < 1 day.

The access log formatting variables follow the Apache variables:

PATTERN	DESCRIPTION
%b	result content length
%D	time taken to complete the request in microseconds (since 3.0.16)
%h	remote IP addr

%{ xxx }i	request header xxx
%{ xxx }o	response header xxx
%{ xxx }c	cookie value xxx
%n	request attribute
%r	request URL
%s	status code
%{ xxx }t	request date with optional time format string.
%T	time taken to complete the request in seconds
%u	remote user
%U	request URI

format patterns

The default format is:

```
"%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\""
```

`resin:type` allows for custom logging. Applications can extend a custom class from . Resin-IoC initialization can be used to set bean parameters in the custom class.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="app-tier">

  <host id="">
    <access-log path='log/access.log'>
      <rollover-period>2W</rollover-period>
    </access-log>
  </host>
</cluster>
</resin>
```

Example: <access-log> in host configuration

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="app-tier">

  <host id='foo.com'>
    <access-log resin:type='test.MyLog'>
                path='${resin.root}/foo/error.log'
                rollover-period='1W'>
      <init>
        <foo>bar</foo>
      </init>
    </access-log>
    ...
  </host>

</cluster>
</resin>
```

Example: custom access log

24.3.2 <ear-deploy>

child of: host, web-app

Specifies ear expansion.

ear-deploy can be used in web-apps to define a subdirectory for ear expansion.

ATTRIBUTE	DESCRIPTION	DEFAULT
archive-path	The path to the directory containing ear files	path
ear-default	resin.xml default configuration for all ear files, e.g. configuring database, JMS or EJB defaults.	
expand-cleanup-fileset	Specifies the files which should be automatically deleted when a new .ear version is deployed.	
expand-directory	directory where ears should be expanded	value of `path`
expand-prefix	automatic prefix of the expanded directory	_ear_
expand-suffix	automatic suffix of the expanded directory	
lazy-init	if true, the ear file is only started on first access	false

path	The path to the deploy directory	required
redeploy-mode	"automatic" or "manual". If automatic, detects new .ear files automatically and deploys them.	automatic
url-prefix	optional URL prefix to group deployed .ear files	

<ear-deploy> Attributes

```
element ear-deploy {
  path
  & archive-directory?
  & ear-default?
  & expand-cleanup-fileset?
  & expand-directory?
  & expand-path?
  & expand-prefix?
  & expand-suffix?
  & lazy-init?
  & redeploy-mode?
  & require-file*
  & url-prefix?
}
```

24.3.3 <error-page>

child of: cluster, host, webapp

<error-page> defines a web page to be displayed when an error occurs outside of a web-app. Note, this is not a default error-page, i.e. if an error occurs inside of a <web-app>, the error-page for that web-app will be used instead.

See webapp: error-page.

24.3.4 <host>

child of: cluster

<host> configures a virtual host. Virtual hosts must be configured explicitly.

ATTRIBUTE	DESCRIPTION	DEFAULT
id	primary host name	none
regexp	Regular expression based host matching	none
host-name	Canonical host name	none
host-alias	Aliases matching the same host	none

secure-host-name	Host to use for a redirect to SSL	none
root-directory	Root directory for host files	parent directory
startup-mode	'automatic', 'lazy', or 'manual', see Startup and Redeploy Mode	automatic

<host> Attributes

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

<host host-name="www.foo.com">
  <host-alias>foo.com</host-alias>
  <host-alias>web.foo.com</host-alias>

  <root-directory>/opt/www/www.foo.com</root-directory>

  <web-app id="/" document-directory="webapps/ROOT">

  </web-app>
  ...
</host>

</cluster>
</resin>
```

Example: explicit host in resin.xml

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

<host regexp="([^.]+)\.foo\.com">
  <host-name>${host.regexp[1]}.foo.com</host-name>

  <root-directory>/var/www/hosts/www.${host.regexp[1]}.com</root-directory>

  ...
</host>

</cluster>
</resin>
```

Example: regexp host in resin.xml

It is recommended that any <host> using a regexp include a <host-name> to set the canonical name for the host.

24.3.5 <host-alias>

<host-alias> defines a URL alias for matching HTTP requests. Any number of <host-alias> can be used for each alias.

The host-alias can be used either in the resin.xml or in a host.xml when use host-deploy together with resin:import.

```
element host-alias {
  string
}
```

```
<resin xmlns="http://caucho.com">
<cluster id="">

  <host id="www.foo.com" root-directory="/var/www/foo.com">
    <host-alias>foo.com</host-alias>

    <web-app id=""/>
  </host>

</cluster>
</resin>
```

Example: host-alias in the resin.xml

Since the <host-deploy> and <host> tags lets you add a host.xml file to customize configuration, the <host-alias> can also fit in the custom host.xml page.

```
<host xmlns="http://caucho.com">

  <host-name>www.foo.com</host-name>
  <host-alias>foo.com</host-alias>

  <web-app id="" root-directory="htdocs"/>

</host>
```

Example: host-alias in a /var/www/hosts/foo/host.xml

24.3.6 <host-alias-regexp>

<host-alias-regexp> defines a regular expression for matching URLs for a given virtual host.

```
element host-alias-regexp {
  string
}
```

```
<resin xmlns="http://caucho.com">
<cluster id="">

  <host id="www.foo.com" root-directory="/var/www/foo.com">
    <host-alias-regexp>.*foo.com</host-alias-regexp>

    <web-app id=""/>
  </host>

</cluster>
</resin>
```

Example: host-alias-regexp in the resin.xml

24.3.7 <host-default>

child of: cluster

<host-default> configures defaults for a virtual host.

The host-default can contain any of the host configuration tags. It will be used as defaults for any virtual host.

24.3.8 <host-deploy>

child of: cluster

<host-deploy> configures an automatic deployment directory for virtual host.

ATTRIBUTE	DESCRIPTION	DEFAULT
archive-directory	path to the archive directory	path
path	path to the deploy directory	required
expand-cleanup-fileset	an ant-style fileset defining which directories to cleanup when an archive is redeployed	
expand-directory	path to the expansion directory	path
host-default	defaults for the expanded host	
host-name	the default hostname, based on the directory	${name}

<host-deploy> Attributes

```
element host-deploy {
  archive-directory?
  & expand-cleanup-fileset?
  & expand-directory?
  & host-default?
  & host-name?
  & path?
}
```

The following example configures `/var/www/hosts` as a host deployment directory. Each virtual host will have a `webapps` directory for .war deployment. So the directory `/var/www/hosts/www.foo.com/webapps/bar/test.jsp` would serve the URL `http://www.foo.com/bar/test.jsp` .

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">
    <root-directory>/var/www</root-directory>

    <host-deploy path="hosts">
      <host-default>
        <resin:import path="host.xml" optional="true"/>

        <web-app-deploy path="webapps"/>
      </host-default>
    </host-deploy>
  </cluster>
</resin>
```

<host-deploy>

24.3.9 <host-name>

<host-name> defines the canonical name for a virtual host. The <host-name> will be used in Resin's logging, management, and is available in the host's variables.

```
element host-name {
  string
}
```

24.3.10 <redeploy-mode>

<redeploy-mode> configures the virtual-host's behavior when it detects changes in configuration files or classes. The <dependency-check-interval> controls how often the virtual host will check for updates.

MODE	DESCRIPTION

automatic	automatically restart when detecting changes
manual	only restart only on a JMX administration request

startup-mode values

```
element startup-mode {
  string
}
```

24.3.11 Resources

child of: resin, cluster, host, web-app

All Resource tags are available to the <host>, for example, resources like <database> or <authenticator>. Resources defined at the host level are available for all web-apps in the host.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">
     <server id="a" .../>

     <host id="www.foo.com">
        <database jndi-name="jdbc/test">
            <driver type="org.postgresql.Driver">
                <url>jdbc:postgresql://localhost/test</url>
                <user>caucho</user>
            </driver>
        </database>

        <web-app-default path="webapps"/>
    </host>
  </cluster>
</resin>
```

Example: shared database in host

24.3.12 <rewrite-dispatch>

child of: cluster, host, web-app

<rewrite-dispatch> defines a set of rewriting rules for dispatching and forwarding URLs. Applications can use these rules to redirect old URLs to their new replacements.

See rewrite-dispatch for more details.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">

    <host host-name="www.foo.com">
      <rewrite-dispatch>
        <redirect regexp="^/foo" target="/index.php?foo="/>
      </rewrite-dispatch>
    </host>

  </cluster>
</resin>
```

rewrite-dispatch

24.3.13 <root-directory>

<root-directory> configures the virtual host's filesystem root.

Because the virtual host's root will typically contain non-public files like log files, all web-apps should have a path below the host.

```
element root-directory {
  string
}
```

24.3.14 <secure-host-name>

<secure-host-name> sets a host-name or URL to be used for secure redirection. For some security configurations, Resin needs to redirect from an insecure site to a secure one. The <secure-host-name> configures the host to redirect to.

See Resin security.

```
element secure-host-name {
  string
}
```

24.3.15 <startup-mode>

<startup-mode> configures the virtual-host's behavior on Resin startup, either "automatic", "lazy" or "manual".

MODE	DESCRIPTION
automatic	automatically start when Resin starts
lazy	start only when the first request is received

manual	start only when JMX administration requests a start

startup-mode values

```
element startup-mode {
  string
}
```

24.3.16 <web-app>

child of: host, web-app

<web-app> configures a web application.

ATTRIBUTE	DESCRIPTION	DEFAULT
id	The url prefix selecting this application.	n/a
url-regexp	A regexp to select this application.	n/a
document-directory	The document directory for the application, corresponding to a url of `/id/` . A relative path is relative to the of the containing . Can use regexp replacement variables.	A relative path constricted with the id or the regexp match
startup-mode	'automatic', 'lazy', or 'manual', see Startup and Redeploy Mode	automatic
redeploy-mode	'automatic' or 'manual', see Startup and Redeploy Mode	automatic

<web-app> Attributes

When specified by `id` , the application will be initialized on server start. When specified by `url-regexp` , the application will be initialized at the first request. This means that `load-on-startup` servlets may start later than expected for `url-regexp` applications.

The following example creates a web-app for /apache using the Apache htdocs directory to serve pages.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="app-tier">

<host id=''>
  <web-app id='/apache' root-directory='/usr/local/apache/htdocs'>

  ...

</host>

</cluster>
</resin>
```

Example: custom web-app root

The following example sets the root web-app to the IIS root directory.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="app-tier">
<host id=''>

  <web-app id='/' root-directory='C:/inetpub/wwwroot'>

</host>
</cluster>
</resin>
```

Example: IIS root directory

When the `web-app` is specified with a `url-regexp` , `root-directory` can use replacement variables (`$2`).

In the following, each user gets his or her own independent application using `user` .

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="app-tier">

  <host id=''>

    <web-app url-regexp='/~([^/]*)'
             root-directory='/home/$1/public_html'>

    ...

    </web-app>

</host>

</cluster>
</resin>
```

Example: web-app root based on regexps

24.3.17 <web-app-default>

child of: cluster, host, web-app

<web-app-default> configures common values for all web applications.

24.3.18 <web-app-deploy>

child of: host, web-app

Specifies war expansion.

web-app-deploy can be used in web-apps to define a subdirectory for war expansion. The tutorials in the documentation use web-app-deploy to allow servlet/tutorial/helloworld to be an independent war file.

ATTRIBUTE	DESCRIPTION	DEFAULT
archive-directory	directory containing the .war files	value of `path`
expand-cleanup-fileset	defines the files which should be automatically deleted when an updated .war expands	all files
expand-directory	directory where wars should be expanded	value of `path`
expand-prefix	prefix string to use when creating the expansion directory, e.g. _war_	
expand-suffix	prefix string to use when creating the expansion directory, e.g. .war	
path	The path to the webapps directory	required
redeploy-check-interval	How often to check the .war files for a redeploy	60s
redeploy-mode	"automatic" or "manual"	automatic
require-file	additional files to use for dependency checking for auto restart	
startup-mode	"automatic", "lazy" or "manual"	automatic
url-prefix	url-prefix added to all expanded webapps	""
versioning	if true, use the webapp's numeric suffix as a version	false

web-app-default	defaults to be applied to expaned web-apps
web-app	overriding configuration for specific web-apps

<web-app-deploy> Attributes

```
element web-app-deploy {
  archive-directory?
  & expand-cleanup-fileset?
  & expand-directory?
  & expand-prefix?
  & expand-suffix?
  & path?
  & redeploy-check-interval?
  & redeploy-mode?
  & require-file*
  & startup-mode?
  & url-prefix?
  & versioning?
  & web-app-default*
  & web-app*
}
```

Overriding web-app-deploy configuration

The web-app-deploy can override configuration for an expanded war with a matching <web-app> inside the <web-app-deploy>. The <document-directory> is used to match web-apps.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">
<host id="">

<web-app-deploy path="webapps">
  <web-app context-path="/wiki"
             document-directory="wiki">
    <context-param database="jdbc/wiki">
  </web-app>
</web-app-deploy>

</host>
</cluster>
</resin>
```

Example: resin.xml overriding web.xml

versioning

The versioning attribute of the <web-app-deploy> tag improves web-app version updates by enabling a graceful update of sessions. The web-apps are named with numeric suffixes, e.g. foo-10, foo-11, etc, and can be browsed as /foo. When a new version of the web-app is deployed, Resin continues to send current session requests to the previous web-app. New sessions go to the new web-app version. So users will not be aware of the application upgrade.

24.4 port: Port tag configuration

See Also

- See the index for a list of all the tags.
- See <cluster> tag configuration
- See <server> tag configuration

24.4.1 <accept-listen-backlog>

child of: http, connection-port, protocol

<accept-listen-backlog> configures operating system TCP listen queue size for the port.

24.4.2 <accept-thread-max>

child of: http, connection-port, protocol

<accept-thread-min> configures the maximum number of threads listening for new connections on this port.

24.4.3 <accept-thread-min>

child of: http, connection-port, protocol

<accept-thread-min> configures the minimum number of threads listening for new connections on this port.

24.4.4 <address>

child of: server

The <address> defines the IP interface for a port. A value of '*' binds to all ports. Because the <address> is specific to a server, it should only be defined in a <server> tag, not a <server-default>.**default:** *

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server id="web-a" address="192.168.1.1" port="6800">
      <http address="10.0.1.1" port="80"/>
    </server>

    <server id="web-b" address="192.168.1.2" port="6800">
      <http address="10.0.1.2" port="80"/>
    </server>

    ...
  </cluster>
</resin>
```

http address

24.4.5 <ca-certificate-file> (OpenSSL)

child of: http, protocol, cluster-port

<ca-certificate-file> configures the path to a CA certificate file for intermediate CA support.

24.4.6 <ca-certificate-path> (OpenSSL)

child of: http, protocol, cluster-port

<ca-certificate-path> configures the path to a CA certificate directory for intermediate CA support.

24.4.7 <ca-revocation-file> (OpenSSL)

child of: http, protocol, cluster-port

<ca-revocation-file> configures the path to a list of revoked CA certificates.

24.4.8 <ca-revocation-path> (OpenSSL)

child of: http, protocol, cluster-port

<ca-revocation-path> configures the path to a list of revoked CA certificates.

24.4.9 <certificate-file> (OpenSSL)

child of: http, protocol, cluster-port

<certificate-file> configures the path to the server's SSL certificate.

24.4.10 <certificate-chain-file> (OpenSSL)

child of: http, protocol, cluster-port

<certificate-chain-file> configures the path to the server's SSL certificate for OpenSSL.

24.4.11 <certificate-key-file> (OpenSSL)

child of: http, protocol, cluster-port

<certificate-key-file> configures the path to the server's SSL private key certificate for OpenSSL.

24.4.12 <cipher-suite> (OpenSSL)

child of: http, protocol, cluster-port

<cipher-suite> configures the path to the server's SSL cryptographic ciphers.

24.4.13 <cluster-port>

child of: server

<cluster-port> configures the cluster and load balancing socket, for load balancing, distributed sessions, and distributed management.

When configuring Resin in a load-balanced cluster, each Resin instance will have its own <srun> configuration, which Resin uses for distributed session management and for the load balancing itself.

When configuring multiple JVMs, each <srun> will have a unique <server-id> which allows the -server command-line to select which ports the server should listen to.

address	hostname of the interface to listen to	*
jsse-ssl	configures the port to use JSSE for SSL	none
openssl	configures the port to use OpenSSL	none
port	port to listen to	required
socket-timeout	timeout waiting to read/write to idle client	65s
accept-listen-backlog	The socket factory's listen backlog for receiving sockets	100
tcp-no-delay	sets the NO_DELAY socket parameter	true

The class that corresponds to <srun> is

24.4.14 <connection-max>

child of: server

<connection-max> configures the maximum number of concurrent connections on this port.

24.4.15 <http>

child of: server

<http> configures a HTTP or HTTPS port listening for HTTP requests.

When configuring multiple JVMs, each <http> will have a unique <server-id> which allows the -server command-line to select which ports the server should listen to.

address	IP address of the interface to listen to	*
port	port to listen to	required
tcp-no-delay	sets the NO_DELAY socket parameter	true
socket-timeout	timeout waiting to write to idle client	65s
accept-listen-backlog	The socket factory's listen backlog for receiving sockets	100
virtual-host	forces all requests to this <http> to use the named virtual host	none
openssl	configures the port to use OpenSSL	none
jsse-ssl	configures the port to use JSSE for SSL	none

The `virtual-host` attribute overrides the browser's Host directive, specifying the explicit host and port for `request.getServerName()` and `getServerPort()` . It is not used in most virtual host configurations. Only IP-based virtual hosts which wish to ignore the browser's Host will use @virtual-host.

24.4.16 <jsse-ssl>

child of: http, protocol, cluster-port

<jsse-ssl> configures the port to use JSSE for SSL support.

The SSL section of the Security documentation provides a comprehensive overview of SSL.

alias	Configures the key alias name in the key store file.	optional
key-store-file	Path to the certificate key store file	required
password	Private key password	required
key-store-type	Type of the keystore	jks
key-manager-factory	Special factory for creating keys	required
ssl-context	Special configuration for the ssl context.	optional
verify-client	Settings for client validation	required

24.4.17 <keepalive-max>

child of: http, connection-port, protocol

<keepalive-max> configures the maximum number of keepalives on this port.

24.4.18 <openssl>

child of: http, protocol, cluster-port

<openssl> configures the port to use OpenSSL for SSL support (requires Resin Professional). OpenSSL is a fast C implementation of SSL security used by Apache. Resin's configuration is OpenSSL follows Apache's configuration, so any documentation on installing SSL certificates can use documentation for Apache.

The SSL section of the Security documentation provides a comprehensive overview of SSL.

ca-certificate-file	Path to a CA certificate file for intermediate CA support	optional
ca-certificate-path	Path to a directory of CA certificates for intermediate CA support	optional
ca-revocation-file	Path to a list of revoked CA certificates	optional
ca-revocation-path	Path to a directory of revoked CA certificates	optional
certificate-file	Path to the server's SSL certificate	required
certificate-chain-file	Path to the certificate chains for client validation.	optional
certificate-key-file	Path to the server's SSL private key certificate	required
cipher-suite	Additions and restrictions to the allowed cryptography ciphers	see openssl-tags
password	Password protecting the public key	see openssl-tags
protocol	Optional restrictions on the SSL protocol	see openssl-tags
session-cache	Boolean enabling caching of SSL sessions for performance	false

session-cache-timeout	Timeout for session cache values	30s
unclean-shutdown	Flag indicating that openssl sockets can be shutdown uncleanly	false
verify-client	Options for client validation	none
verify-depth	Depth of the client certificate chains to validate	unlimited

24.4.19 <password> (OpenSSL)

child of: openssl

<password> configures the SSL private key certificate password.

24.4.20 <port>

child of: http, protocol, server

The <port> defines the TCP port the HTTP or protocol should bind to.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server id="web-a" address="192.168.1.1" port="6800">
      <http address="10.0.1.1" port="80"/>
      <http address="192.168.1.1" port="8080"/>
    </server>

    ...
  </cluster>
</resin>
```

http port

24.4.21 <protocol>

child of: server

<protocol> configures custom socket protocols using Resin's thread and connection management.

The custom protocol will extend from com.caucho.server.port.Protocol.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="web-tier">

  <server id="a">
    <protocol address="localhost" port="8888">
      <type>example.Magic8BallProtocol</type>
    </port>
  </server>

</cluster>
</resin>
```

24.4.22 <protocol> (OpenSSL)

child of: openssl

<protocol> configures the SSL allowed protocols.

24.4.23 <session-cache> (OpenSSL)

child of: openssl

<session-cache> configures the SSL session cache.

24.4.24 <session-cache-timeout> (OpenSSL)

child of: openssl

<session-cache> configures the SSL session cache timeout.

24.4.25 <socket-timeout>

child of: http, cluster-port, protocol, server

<socket-timeout> overrides the socket timeout from the <server>.

24.4.26 <tcp-no-delay>

child of: http, protocol, cluster-port

Sets the tcp-no-delay parameter.

24.4.27 <unclean-shutdown> (OpenSSL)

child of: openssl

<unclean-shutdown> configures the OpenSSL unclean shutdown on connection close.

24.4.28 <verify-client> (OpenSSL)

child of: openssl

<verify-client> sets the client certificate configuration. If the certificate is available, it will be put in the javax.servlet.request.X509Certificate request attribute.

none	do not ask for a client certificate (default)
required	require a client certificate
optional	ask for a client certificate if availiable
optional-no-ca	ask for a client certificate, but do not validate the Certificate Authority

```
X509Certificate []certs = (X509Certificate [])
     request.getAttribute("javax.servlet.request.X509Certificate");
```

Obtaining the client certificate

24.4.29 <verify-depth> (OpenSSL)

child of: openssl

<verify-depth> configures the OpenSSL client verification depth.

24.5 Resources: class loaders, environment and IoC

24.5.1 <authenticator>

child of: resin, cluster, host, web-app, login-config

<authenticator> configures an authentication resource for the current environment context. The authenticator is used for login and also for the `getUserPrincipal` and `isUserInRole` methods of the `HttpServletRequest` object.

The authenticators are scoped to their containing environment. An authenticator defined in WEB-INF/resin-web.xml applies only to the web-app, while an authenticator defined in the <cluster> section of the resin.xml applies to the entire cluster. The <management> configuration provides an authenticator which is available to all applications.

Resin's servlet authentication uses an authentication resource to validate user login and to provide single-signon capability. The authenticator is configured in the environment context where it is shared. An authenticator configured

in the web-app only applies to the web-app, but an authenticator configured in the host will apply to all hosts.

The authenticator class is selected with the `uri` or `class` attribute. The `class` can be any custom class extending `com.caucho.server.security.AbstractAuthenticator` . The `uri` is a symbolic name for the authenticator class. More details on the predefined authenticators are in the Resin security documentation.

- `properties:` Java properties-style authentication.
- `jaas:` JAAS authentication.
- `jdbc:` JDBC password-based authentication.
- `xml:` XML JDBC password-based authentication.

Configuration of the authenticator uses bean-style configuration in the <init> tag.

See also: the Resin security section.

ATTRIBUTE	DESCRIPTION	DEFAULT
binding	Custom WebBeans binding. Since Resin 3.1.5	
class	The implementing class for the authenticator. Since Resin 3.0	
jndi-name	The JNDI name where the authenticator is stored. Since Resin 3.0	
name	The Resin-IoC name where the authenticator is stored. Since Resin 3.1.5	
uri	shortcut alias for the authenticator class. Can also include inline parameters. Examples include xml:, property:, jdbc:, jndi:.	
init	A bean-style configuration section. Since Resin 3.0	

<authenticator> attributes

```
r_authenticator = element authenticator {
  (r_class | r_uri)
  & r_binding*
  & r_init?
  & r_jndi-name?
  & r_name?
  & r_scope?
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

  <authenticator uri="properties:password-digest=none">
    <init>
harry=quidditch,user
ron=cannons,user,prefect
    </init>
  </authenticator>

</web-app>
```

Example: WEB-INF/resin-web.xml properties-based authenticator

24.5.2 <bam-service>

child of: resin, cluster, host, web-app

<bam-service> configures a BamService to listen for messages. The service is always a single instance.

<bam-service> can be used as a simple queueing service, replacing JMS queues and ejb-message-bean. By default the queue is consumed with a single thread. Unlike EJB message driven beans, <bam-service> uses a single instance like a servlet, not a pool of instances.

The bean has full access to Resin-IoC capabilities, including dependency injection, transaction attributes, and aspect interception.

ATTRIBUTE	DESCRIPTION	DEFAULT
class	Classname of the BamService bean	required
init	IoC configuration for the listener bean	
name	The JID of the service, used by clients to send messages to	
thread-max	The number of threads to handle the queue. If 0, the service is not queued, but handles messages directly.	1

uri	A shortcut name for the service, defined in META-INF/caucho/com.caucho.	Service.

<bam-service> attributes

```
element ejb-message-bean {
  class
  & init?
  & name?
  & uri?
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

  <bam-service uri="caucho.log:">
    <name>log</name>
    <init>
      <name>test.Log<name>
    </init>
  </bam-service>

</web-app>
```

Example: Log service in WEB-INF/resin-web.xml

24.5.3 <bean>

child of: resin, cluster, host, web-app

<bean> configures a custom singleton bean and stores in the WebBeans registry. <bean> is a primary configuration tag for Resin's IoC capabilities. The bean can also be optional registered in JNDI.

Custom configuration of the bean is in the <init> section. Field values may use JSP-EL expressions as well as constant strings or even complex sub-beans. More details for configuring singleton beans are in Resin IoC.

ATTRIBUTE	DESCRIPTION	DEFAULT
class	Application class implementing the resource. Since Resin 3.0	required
init	IoC configuration for the bean	
jndi-name	JNDI name for the resource. Since Resin 3.0	

mbean-name	JMX name for management registration. Resin 3.0	
name	The name of the bean, used for @Named injection. Resin 3.1.4.	
scope	request, session, conversational, application, singleton. Resin 3.1.4.	singleton

<bean> attributes

```
r_bean = element bean {
  r_class?
  & r_binding*
  & r_init?
  & r_jndi-name?
  & r_mbean-name?
  & r_mbean-interface?
  & r_name?
  & r_scope?
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">
  <bean name="test">
    <type>test.MyBean</type>
    <init>
      <greeting>Hello</greeting>
      <server>${serverId}</server>
      <sub-bean>
        <value>${2 + 2}</value>
      </sub-bean>
    </init>
  </bean>
</web-app>
```

Example: WEB-INF/resin-web.xml singleton bean

24.5.4 <case-insensitive>

child of: resin, cluster, host, web-app**default:** true on Windows, false on Unix.

<case-insensitive> specifies whether the environment context is case sensitive or insensitive.

Because some operating systems are case-insensitive, it is important for security reasons for Resin to behave differently for case-sensitive and case-insensitive directories. For example, when case-insensitive is true, url-patterns will match in a case-insensitive manner, so TEST.JSP will work like test.jsp.

```
r_case-insensitive = element case-insensitive {
  r_boolean-Type
}
```

24.5.5 <character-encoding>

child of: resin, cluster, host, web-app**default:** The default value is ISO-8859-1.

<character-encoding> specifies the default character encoding for the environment.

```
r_character-encoding = element character-encoding {
  string
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
  <character-encoding>utf-8</character-encoding>
  ...

</resin>
```

Example: utf-8 as default character encoding

24.5.6 <class-loader>

child of: resin, cluster, host, web-app

<class-loader> configures a dynamic classloader for the current environment.

Each environment (<cluster>, <host>, <web-app>) etc, can add dynamic classloaders. The environment will inherit the parent classloaders. Each <class-loader> is comprised of several implementing loader items: library-loader for WEB-INF/lib, compiling-loader for WEB-INF/classes.

For web-apps, the classloaders generally belong in a <prologue> section, which ensures that Resin evaluates them first. The evaluation order is particularly important in cases like resin-web.xml vs web.xml, because the resin-web.xml is evaluated after the web.xml.

ELEMENT	DESCRIPTION
<compiling-loader>	Automatically compiles sources code to classes. It its the default loader for WEB-INF/classes.
<library-loader>	Loads jar files from a directory. It is the default loader for WEB-INF/lib.
<simple-loader>	Loads classes from a directory, but does not compile them automatically.
<tree-loader>	Loads jar files from a directory, recursively searching subdirectories.

classloader types

```
r_class-loader = element class-loader {
  r_compiling-loader*

  & r_library-loader*

  & r_simple-loader*

  & r_tree-loader*
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">
  <prologue>
    <class-loader>
      <compiling-loader path="WEB-INF/classes"/>

      <library-loader path="WEB-INF/lib"/>
    </class-loader>
  </prologue>
</web-app>
```

Example: WEB-INF/resin-web.xml defined <class-loader>

24.5.7 <compiling-loader>

child of: class-loader

<compiling-loader> automatically compiles Java code into .class files before loading them.

ATTRIBUTE	DESCRIPTION	DEFAULT
args	Additional arguments to be passed to the Java compiler. Resin 3.0	
batch	If true, multiple changed *.java files will be compiled in a single batch. Resin 3.0.7	true
encoding	I18N encoding for the Java compiler. Since Resin 3.0	
path	Filesystem path for the class loader. Since Resin 3.0	required
source	Java source directory. Since Resin 3.0	value of path
require-source	If true, .class files without matching .java files will be deleted. Since Resin 3.0	false

<compiling-loader> attributes

```
<web-app xmlns="http://caucho.com/ns/resin">
  <prologue>
    <class-loader>
      <compiling-loader path="WEB-INF/classes"
                        source="WEB-INF/src"/>
    </class-loader>
  </prologue>
</web-app>
```

Example: WEB-INF/resin-web.xml <compiling-loader>

24.5.8 <component>

child of: resin, cluster, host, web-app

<component> configures a component bean template and stores in the WebBeans registry. Injection of a <component> will generally create a new bean instance in constrast to the singleton <bean>. <component> is a primary configuration tag for Resin's IoC capabilities. The bean can also be optionally registered in JNDI.

Custom configuration of the component is in the <init> section. Field values may use JSP-EL expressions as well as constant strings or even complex sub-beans. More details for configuring singleton beans are in Resin IoC.

ATTRIBUTE	DESCRIPTION	DEFAULT
class	Application class implementing the resource. Since Resin 3.0	required
init	IoC configuration for the bean	
jndi-name	JNDI name for the resource. Since Resin 3.0	
name	The name of the bean, used for @Named injection. Resin 3.1.4.	
scope	dependent, request, session, conversational, application, singleton. Resin 3.1.4.	dependent

<component> attributes

```
r_component = element component {
  r_class?
  & r_binding*
  & r_init?
  & r_jndi-name?
  & r_name?
  & r_scope?
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

  <component name="test">
    <type>test.MyBean</type>
    <init>
      <greeting>Hello</greeting>
      <server>${serverId}</server>
      <sub-bean>
        <value>${2 + 2}</value>
      </sub-bean>
    </init>
  </component>

</web-app>
```

Example: WEB-INF/resin-web.xml component bean

24.5.9 <connection-factory>

child of: resin, cluster, host, web-app

JDBC CLASS	CONNECTOR CLASS
Connection	Connection (driver-defined class)
DataSource	ConnectionFactory (driver-defined class)
Driver	ManagedConnectionFactory
n/a	ResourceAdapter

Parallels to JDBC

ATTRIBUTE	DESCRIPTION	DEFAULT
class	ManagedConnectionFact driver for the resource	required or `uri`
init	IoC configuration for the ManagedConnectionFactory	
local-transaction-optimization	Enables the local transaction optimization during commit	true
max-active-time	Configures the maximum time allowed for a connection	infinite
max-connections	Configures the maximum connections available	1024

jndi-name	JNDI name for the `ConnectionFactory`	
name	The IoC name for the `ConnectionFactory`, used for @Named injection. Resin 3.1.4.	
resource-adapter	The driver's `ResourceAdapter`	
shareable	Enables sharing of `Connection` objects in a transaction	true
uri	Shortcut alias for the ManagedConnection-Factory	required or `class`

<connection-factory> attributes

```
r_connection-factory = element connection-factory {
  (r_class | r_uri)
  & r_binding*
  & r_init?
  & r_jndi-name?
  & r_name?

  & local-transaction-optimization?
  & max-active-time?
  & max-connections?
  & resource-adapter?
  & shareable?
}
```

24.5.10 <database>

child of: resin, cluster, host, web-app

<database> defines a database (i.e. DataSource) resource.

The database configuration section has more details on the configuration. A code pattern for using databases is in a DataSource tutorial.

ATTRIBUTE	DESCRIPTION	DEFAULT
backup-driver	Configures a backup database driver. If Resin can't connect to any of the main drivers, it will use one of the. backups	
close-dangling-connections	If an application does not close a Connection by the end of the request, Resin will close it automatically an issue a warning.	true
connection	Defines initialization attributes for new connections, e.g. setting the transaction-isolation.	true
connection-wait-time	When max-connections has been reached, how long Resin will wait for a connection to become idle before giving up.	10min
driver	Configures the database driver, giving the driver's class name as well as its JDBC URL and any other configuration.	required
jndi-name	The JNDI name to register the connection's `DataSource` under. If the name can be relative to `java:comp/env`.	
max-active-time	The maximum time Resin will allow a connection to remain open before forcing a close.	6 hours

max-close-statements	The maximum number of Statements Resin will hold to automatically close when the Connection closes.	256
max-connections	The maximum number of Connections allowed.	128
max-create-connections	The maximum number of connection creation allowed at one time.	5
max-idle-count	The maximum number of Connections in the idle pool.	1024
max-idle-time	The maximum time a connection will spend in the idle pool before closing.	30s
max-overflow-connections	The number of extra connection creation if the number of connections exceeds to pool size.	0
max-pool-time	The total time a connection can be used before it is automatically closed instead of returned to the idle pool.	24h
name	The IoC name to save the `ConnectionFactory` as, used with `@Named` to inject the resource.	
password	The JDBC password for the connection.	
ping	If true, Resin will ping the database before returning a connection from the pool (if ping-interval is exceeded).	false
ping-interval	How often an idle connection should ping the database to ensure it is still valid.	1s
ping-query	A custom query used to ping the database connection.	

ping-table	A table used to ping the database connection.	
prepared-statement-cache-size	How many `PreparedStatements` to save in the prepared statement cache.	0
save-allocation-stack-trace	If true, saves the location of the connection allocation as a stack trace.	false
spy	Enables spy logging of database statements. The logging occurs with name="com.caucho.sql" and level="fine".	false
transaction-timeout	Sets the transaction timeout.	none
user	Sets the authentication user.	
wrap-statements	If true, Resin wraps statements and automatically closes them on connection close.	true
xa	Enables automatic enlistment of `Connections` with any `UserTransaction`. Disabling <xa> means the connection are independent of transactions, useful for read-only connections.	true
xa-forbid-same-rm	Workaround flag to handle certain database drivers that do not properly implement the XAResource API.	false

<database> attributes

```
database = element database {
  backup-driver*
  & close-dangling-connections?
  & connection?
  & connection-wait-time?
  & driver+
  & jndi-name?
  & max-active-time?
  & max-close-statements?
  & max-connections?
  & max-create-connections?
  & max-idle-count?
  & max-idle-time?
  & max-overflow-connections?
  & max-pool-time?
  & name?
  & password?
  & ping?
  & ping-interval?
  & ping-query?
  & ping-table?
  & prepared-statement-cache-size?
  & save-allocation-stack-trace?
  & spy?
  & transaction-timeout?
  & user?
  & wrap-statements?
  & xa?
  & xa-forbid-same-rm?
}

backup-driver = element backup-driver {
  class?
  & url?
  & element * { * }?
}

connection = element connection {
  catalog?
  & read-only?
  & transaction-isolation?
}

driver = element driver {
  class?
  & url?
  & element * { * }?
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

<database jndi-name='jdbc/test_mysql'>
  <driver class="com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource">
    <url>jdbc:mysql://localhost:3306/test</url>
    <user></user>
    <password></password>
  </driver>
</database>

</web-app>
```

Example: WEB-INF/resin-web.xml database

24.5.11 <database-default>

child of: resin, cluster, host, web-app

<database-default> defines default database values to be used for any <database> definition, or runtime database creation (see DatabaseManager).

```
element database-default {
  r_database-Content
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

  <database-default>
    <max-idle-time>10s</max-idle-time>
  </database-default>

</web-app>
```

Example: WEB-INF/resin-web.xml idle-time defaults

24.5.12 <dependency>

child of: resin, cluster, host, web-app

<dependency> adds dependent files which should force a reload when changed, like web.xml and resin-web.xml.

ATTRIBUTE	DESCRIPTION	DEFAULT
path	Filesystem path to the dependent file. Since Resin 3.0	required

<dependency> attributes

```
element dependency {
  string
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">
  <dependency path="WEB-INF/struts-config.xml"/>
  ...
</web-app>
```

Example: struts dependency

24.5.13 <dependency-check-interval>

child of: resin, cluster, host, web-app**default:** 2s

<dependency-check-interval> Configures how often the environment context should be checked for changes. The default value is set low for development purposes, deployments should use something larger like 5m or 1h.

Resin automatically checks each environment for updates, generally class or configuration updates. Because these checks can take a considerable amount of time, deployment servers should use high values like 60s or more while development machines will want low values like 2s.

The interval defaults to the parent's interval. So the web-app will default to the host's value.

```
element dependency-check-interval {
  string
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">
    <dependency-check-interval>1h<dependency-check-interval>

    <server id="app-a" .../>

    <host id=""/>
      ...
  </cluster>
</resin>
```

Example: deployment dependency-check-interval

24.5.14 <ejb-message-bean>

child of: resin, cluster, host, web-app

<ejb-message-bean> configures a bean as a message listener. The listener can be a simple bean that just implements the `javax.jms.MessageListener interface` . No other packaging or complications are necessary. Resin will retrieve messages from a configured queue and pass them to the listener as they arrive. The listeners are typically pooled.

The bean has full access to Resin-IoC capabilities, including dependency injection, transaction attributes, and aspect interception.

The message bean can plug into custom messaging systems. The application will need to define a ResourceAdapter and an `ActivationSpec` . More details are available in the Resin Messaging section.

ATTRIBUTE	DESCRIPTION	DEFAULT
activation-spec	Configures a custom message-listener driver	

class	Classname of the listener bean	required
destination	Queue or Topic for JMS message receiving	
destination-type	javax.jms.Queue or javax.jms.Topic	
init	IoC configuration for the listener bean	
message-consumer-max	The number of listener instances to create for the pool.	5

<ejb-message-bean> attributes

```
element ejb-message-bean {
  class
  & init?
  & (activation-spec?
     | (destination?
        & destination-type?
        & destination-name?
        & message-consumer-max?)
    )
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

  <jms-connection-factory uri="resin:"/>
  <jms-queue name="my_queue" uri="memory:"/>

  <ejb-message-bean class="qa.MyListener">
    <destination>${my_queue}</destination>
  </ejb-message-bean>

</web-app>
```

Example: JMS listener in WEB-INF/resin-web.xml

```
<web-app xmlns="http://caucho.com/ns/resin">

  <resource-adapter uri="activemq:"/>

  <ejb-message-bean class="qa.MyListener">
    <activation-spec uri="activemq:">
       <init physical-name="queue.test"/>
    </activation-spec uri="activemq:">
  </ejb-message-bean>

</web-app>
```

Example: ActiveMQ in WEB-INF/resin-web.xml

24.5.15 <ejb-server>

child of: resin, cluster, host, web-app

Configures an EJB server. See Resin EJB for more details.

ATTRIBUTE	DESCRIPTION	DEFAULT
auto-compile	enables auto-compilation of EJB stubs and skeletons	true
create-database-schema	enables JPA auto-creation of missing database tables	false
data-source	specifies the default database for JPA	
config-directory	specifies a directory containing *.ejb configuration files	
ejb-descriptor	path to a *.ejb file to load	
ejb-jar	path to a jar file containing a META-INF/ejb-jar.xml with EJBs	
jndi-prefix	prefix for JNDI registration of EJBs	
validate-database-schema	verifies the actual database tables against the JPA definitions	true
jms-connection-factory	specifies the default JMS ConnectionFactory for message beans	

xa-data-source	specifies a separate database for transactions	data-source

<ejb-server> attributes

```
element ejb-server {
  auto-compile
  & create-database-schema
  & data-source
  & config-directory
  & ejb-descriptor
  & ejb-jar
  & jndi-prefix
  & validate-database-schema
  & jms-connection-factory
  & xa-data-source
}
```

24.5.16 <ejb-stateful-bean>

child of: resin, host-default, host, web-app-default, web-app

<ejb-stateful-bean> configures an EJB @Stateful bean. The @Stateful bean is a single-threaded component bean suitable for transaction processing. See Resin EJB for more details.

The stateful-bean is registered in the Resin-IoC/WebBeans context and optionally with JNDI.

Since @Stateful beans are components, they are created at the request of the application and destroyed by the application. @Stateful beans are never singletons. For singleton-style beans, either use a <bean> or a @Stateless session bean.

@Stateful beans may optionally implement a SessionSynchronization interface for transaction callbacks.

ATTRIBUTE	DESCRIPTION	DEFAULT
class	the classname of the bean implementation	required
init	IoC initialization for each bean instance	
jndi-name	A JNDI name to store the bean as.	
name	The Resin-IoC/WebBeans @Named registration	The classname
scope	The Resin-IoC/WebBeans scope: dependent, request, session, conversaion	dependent classname

<ejb-stateful-bean> Attributes

```
element ejb-stateful-bean {
  class
  & init?
  & jndi-name?
  & name?
  & scope?
}
```

24.5.17 <ejb-stateless-bean>

child of: resin, host-default, host, web-app-default, web-app

<ejb-stateless-bean> configures an EJB @Stateless bean. The @Statelesss bean is a pooled, proxied, singleton component bean suitable. See Resin EJB for more details.

The stateless-bean is registered in the Resin-IoC/WebBeans context and optionally with JNDI.

@Stateless beans are similar to <bean> singletons, but pool instances. Each instance executes a single thread at a time, unlike <bean> singletons which are multithreaded like servlets. Both styles can use the same aspect capabilities like dependency injection, transactions, and interceptors. Because @Stateless beans are singletons, they do not have a scope attribute.

ATTRIBUTE	DESCRIPTION	DEFAULT
class	the classname of the bean implementation	required
init	IoC initialization for each bean instance	
jndi-name	A JNDI name to store the bean as.	
name	The Resin-IoC/WebBeans @Named registration	The classname

<ejb-stateless-bean> Attributes

```
element ejb-stateless-bean {
  class
  & init?
  & jndi-name?
  & name?
}
```

24.5.18 <env-entry>

child of: resin, host-default, host, web-app-default, web-app

<env-entry> configures a JNDI scalar value for JNDI-based application configuration.

Some application beans prefer to retrieve configuration data from JNDI, including String, Integer, and Double constants. env-entry configures that data in the current context. As with other Resin configuration, the value can use JSP-EL expressions.

ATTRIBUTE	DESCRIPTION	DEFAULT
env-entry-name	JNDI name to store the value. Since Servlet 2.1	required
env-entry-type	Java type for the value. Since Servlet 2.1	required
env-entry-value	Value to be stored. Since Servlet 2.1	required

<env-entry> attributes

```
element env-entry {
  description*,

  env-entry-name,

  env-entry-type,

  env-entry-value
}
```

The example configuration stores a string in java:comp/env/greeting. Following the J2EE spec, the env-entry-name is relative to java:comp/env. If the env-entry is in the <host> context, it will be visible to all web-apps in the host.

```
<web-app xmlns="http://caucho.com/ns/resin">
  <env-entry>
    <env-entry-name>greeting</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>Hello, World</env-entry-value>
  </env-entry>

  <servlet ...>
  </servlet>
</web-app>
```

Example: WEB-INF/resin-web.xml with env-entry

The following servlet fragment is a typical use in a servlet. The servlet only looks up the variable once and stores it for later use.

```
import java.io.*;
import javax.naming.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class TestServlet extends HttpServlet {
  private String greeting;

  public void init()
    throws ServletException
  {
    try {
      Context env =
        (Context) new InitialContext().lookup("java:comp/env");
      greeting = (String) env.lookup("greeting");
    } catch (NamingException e) {
      throw new ServletException(e);
    }
  }

  ...
}
```

Example: GreetingServlet.java

24.5.19 <fileset>

<fileset> provides the ability to match a set of files. It is modelled after the ant tag by the same name. The fileset matches files from a base directory defined by 'dir'. Files can be included by patterns defined by <include> tags or excluded by patterns defined in <exclude> tags.

ATTRIBUTE	DESCRIPTION	DEFAULT
dir	the starting directory	required
include	an include pattern	do not include all files
exclude	an exclude pattern	do not exclude any files

<fileset> Attributes

```
element fileset {
  dir
  & exclude*
  & include*
```

fileset patterns

A pattern can contain two special characters: '*' and '**'. '*' matches any part of path, but does not match the path separator. '**' matches any part of a path, including the path separator.

The following example matches .jar files in WEB-INF/lib. Since it does not search the lib directory recursively, WEB-INF/lib/foo/bar.jar will not match.

```
<fileset dir="WEB-INF/lib">
  <include name="*.jar"/>
</fileset>

MATCH    lib/foo.jar
MATCH    lib/bar.jar
NO MATCH lib/baz/foo.jar
```

Example: fileset pattern '*'

The following example matches .jar files in WEB-INF/lib recursively, so a deeper file like WEB-INF/lib/foo/bar.jar will match.

```
<fileset dir="WEB-INF/tree">
  <include name="**/*.jar"/>
</fileset>

MATCH    lib/foo.jar
MATCH    lib/bar.jar
MATCH    lib/baz/foo.jar
```

Example: fileset pattern '**'

24.5.20 <javac>

child of: resin, cluster, host, web-app

<javac> configures the Java compiler for automatically compiled files.

The javac configuration is used for JSP, PHP, EJB and compiling-loader configuration.

ATTRIBUTE	DESCRIPTION	DEFAULT
args	extra arguments to pass to the compiler	
compiler	the compiler name: eclipse, groovyc, internal, or a command-line	
encoding	the character encoding to use	utf-8
max-batch	the maximum number of source files to batch into one compilation	64

<javac> Attributes

```
element javac {
  args*
  & compiler
  & encoding?
  & max-batch?
}
```

The eclipse compiler requires the presence of $RESIN_HOME/lib/eclipse-compiler.jar (which is included with Resin). It is a very fast compiler that was developed as part of the Eclipse project.

```
<resin xmlns="http://caucho.com/ns/resin">

 <javac compiler="eclipse" args="-source 1.5"/>

  ...

</resin>
```

Example: eclipse compiler

The internal compiler requires tools.jar from the JDK installation, so a JDK must be used (not a JRE). Sometimes the internal compiler causes errors, creating exceptions or simply hanging and taking up a thread. The solution is to change the compiler to use an external compiler.

```
<resin xmlns="http://caucho.com/ns/resin">

 <javac compiler="internal" args=""/>

</resin>
```

Internal compiler

The javac compiler is included with the JDK. It executes that same as the internal compiler, however it is executed as an external process and is less prone to the problems described for the internal compiler. In resin.xml with the javac configuration option:

```
<resin xmlns="http://caucho.com/ns/resin">

 <javac compiler="javac" args=""/>

  ...

</resin>
```

javac JDK compiler

24.5.21 <jms-connection-factory>

child of: resin, cluster, host, web-app

<jms-connection-factory> configures a JMS `ConnectionFactory` and registers it with Resin-IoC/WebBeans. The `ConnectionFactory` can be specified by it's class or a URI alias.

See Resin messaging for more information.

SCHEME	DESCRIPTION
resin:	Resin's ConnectionFactory implementation

jms-connection-factory URIs

ATTRIBUTE	DESCRIPTION	DEFAULT
class	class name of the ConnectionFactory implementation	required (or `uri`)
init	IoC initialization configuration for the factory	
jndi-name	Name for registering in JNDI, relative to java:comp/env	
name	@Named binding in Resin-IoC	
uri	URI alias for the ConnectionFactory class	

<jms-connection-factory> Attributes

```
element jms-connection-factory {
  (class | uri)
  & init?
  & jndi-name?
  & name?
}
```

24.5.22 <jms-queue>

child of: resin, cluster, host, web-app

<jms-queue> configures a JMS `Queue` and registers it with Resin-IoC/WebBeans. Resin's JMS Queues implement the `BlockingQueue` interface and can be used directly or with the JMS `ConnectionFactory` .

See Resin messaging for more information.

The Queue can either be identified by the implementing class name or with a URI alias. Third-party queues using JCA will use the <resource-adapter> and either <connection-factory> or <activation-spec>.

Code sending messages to a Queue will typically use either the `BlockingQueue` API or JMS. Code receiving from a queue will generally use an EJB message bean.

ATTRIBUTE	DESCRIPTION	DEFAULT
class	class name of the Queue implementation	required (or `uri`)
init	IoC initialization for the Queue	
jndi-name	JNDI name for storing the Queue	
name	@Named binding for Resin-IoC	

<jms-queue> Attributes

URI	DESCRIPTION
file:	Persistent, file-backed Queue
memory:	Simple, memory-based Queue
server:	Server side of a clustered Queue
client:	Client side of a clustered Queue

Queue URI aliases

```
element jms-queue {
  (class | uri)
  & init
  & jndi-name
  & name
}
```

24.5.23 <jms-topic>

child of: resin, cluster, host, web-app

<jms-topic> configures a JMS `Topic` and registers it with Resin-IoC/WebBeans. Resin's JMS Topics implement the `BlockingTopic` interface and can be used directly or with the JMS `ConnectionFactory` .

See Resin messaging for more information.

The Topic can either be identified by the implementing class name or with a URI alias. Third-party topics using JCA will use the <resource-adapter> and either <connection-factory> or <activation-spec>.

Code sending messages to a Topic will typically use either the `BlockingTopic` API or JMS. Code receiving from a topic will generally use an EJB message bean.

ATTRIBUTE	DESCRIPTION	DEFAULT
class	class name of the Topic implementation	required (or `uri`)
init	IoC initialization for the Topic	
jndi-name	JNDI name for storing the Topic	
name	@Named binding for Resin-IoC	

<jms-topic> Attributes

URI	DESCRIPTION
file:	Persistent, file-backed Topic
memory:	Simple, memory-based Topic
server:	Server side of a clustered Topic
client:	Client side of a clustered Topic

Topic URI aliases

```
element jms-topic {
  (class | uri)
  & init
  & jndi-name
  & name
}
```

24.5.24 <jndi-link>

child of: resin, cluster, host, web-app

<jndi-link> creates a symbolic link from one jndi name to another, or links to a foreign JNDI context.

Resin's JNDI can link to foreign JNDI contexts. For example, third-party EJB servers will often expose their EJB beans through a JNDI context. jndi-link will create the appropriate InitialContextFactory, configure it, and lookup the foreign JNDI objects.

ATTRIBUTE	DESCRIPTION	DEFAULT

factory	Class name of the JNDI InitialContextFactory. Since Resin 1.2	optional
foreign-name	The target name of the symbolic link, or the sub-context of the foreign JNDI context. Since Resin 1.2	none
init-param	Configuration parameters for the JNDI environment passed to InitialContextFactory. Since Resin 1.2	none
jndi-name	The JNDI name to use for the link. Resin 3.0	required

<jndi-link> Attributes

```
element jndi-link {
  jndi-name
  & factory?
  & foreign-name?
  & init-param*
}
```

```
<web-app xmlns="http://caucho.com/ns/resin"dd>
  <database jndi-name="jdbc/oracle">
    ...
  </database>

  <jndi-link jndi-name="java:comp/env/jdbc/gryffindor">
    <foreign-name>java:comp/env/jdbc/oracle</foreign-name>
  </jndi-link>

  <jndi-link jndi-name="java:comp/env/jdbc/slytherin">
    <foreign-name>java:comp/env/jdbc/oracle</foreign-name>
  </jndi-link>
</web-app>
```

Example: A JNDI symbolic link for a DataSource

```
<web-app xmlns="http://caucho.com/ns/resin">
  <jndi-link jndi-name='java:comp/env/ejb'>
    <factory>com.caucho.ejb.hessian.HessianContextFactory</factory>
    <init-param java.naming.provider.url='http://ejb.hogwarts.com:80/hessian'/>
  </jndi-link>
</web-app>
```

Example: A JNDI foreign context for all EJB

```
<web-app xmlns="http://caucho.com/ns/resin">
  <jndi-link jndi-name='java:comp/env/remote-ejb'>
    <factory>com.caucho.ejb.hessian.HessianContextFactory</factory>
    <init-param java.naming.provider.url='http://ejb.hogwarts.com:80/hessian'/>
  </jndi-link>

  <jndi-link jndi-name="java:comp/env/ejb/Foo">
    <foreign-name>java:comp/env/remote-ejb/Foo</foreign-name>
  </jndi-link>

  <jndi-link jndi-name="java:comp/env/ejb/Bar">
    <foreign-name>java:comp/env/local-ejb/Bar</foreign-name>
  </jndi-link>
</web-app>
```

Example: A JNDI foreign context for selected EJB

24.5.25 <jpa-persistence>

child of: resin, cluster, host, web-app

<jpa-persistence> configures JPA persistence defaults. More details on JPA are available on the Amber page.

ATTRIBUTE	DESCRIPTION	DEFAULT
create-database-schema	If true, Amber will automatically create the database schema	false
cache-size	Size of the entity cache	32k
data-source	database used for JTA	
jdbc-isolation	JDBC isolation level used for connections	
read-data-source	Data source to be used for read-only queries	data-source
validate-database-schema	enables validation of the database tables on startup	false
xa-data-source	database to use in transactions	data-source

<jpa-persistence> Attributes

```
element jpa-persistence {
  create-database-schema?
  & cache-size?
  & cache-timeout?
  & data-source?
  & jdbc-isolation?
  & persistence-unit*
  & persistence-unit-default*
  & read-data-source?
  & validate-database-schema?
  & xa-data-source?
}

element persistence-unit {
  name
  & jta-data-source?
  & non-jta-data-source?
  & provider?
  & transaction-type?
  & properties?
}

element persistence-unit-default {
  & jta-data-source?
  & non-jta-data-source?
  & provider?
  & transaction-type?
  & properties?
}

element properties {
  element property {
     name
     & value
  }*
}
```

24.5.26 <library-loader>

child of: class-loader

<library-loader> configures a jar library, `WEB-INF/lib` -style class loader.

The library-loader will add jar files in its path to the current classpath. Jar files are recognized wihen they have a filename extension of `.jar` or `.zip` .

ATTRIBUTE	DESCRIPTION	DEFAULT
fileset	An ant-style fileset	
path	Filesystem path for the class loader. Since Resin 3.0	required

```
element library-loader {
  fileset
  | path
}

element fileset {
  dir
  & exclude*
  & include*
}
```

See DirectoryLoader.

24.5.27 <log>

child of: resin, cluster, host, web-app

<log> configures JDK 1.4 java.util.logger handler.

The log configuration describes log in detail.

ATTRIBUTE	DESCRIPTION	DEFAULT
archive-format	defines a format string for log rollover	
format	defines the output formatting	
formatter	defines a custom formatter	
handler	defines a custom handler	
level	sets the logging level of the handler	info
mbean-name	sets an mbean-name to register the logger for runtime management	
path	sets the VFS path for the log file	
path-format	sets a pattern for creating the VFS path for the messages	
rollover-count	sets the maximum number of rollover files	
rollover-period	sets the number of days before a log file rollover	1m
rollover-size	sets the maximum log size before a rollover	1g
timestamp	sets the formatting string for the timestamp label	

use-parent-handlers	if true, the log is also copied to parent handlers	true

<log> Attributes

```
element log {
  archive-format?
  & format?
  & formatter?
  & handler?
  & level?
  & mbean-name?
  & name
  & path?
  & path-format?
  & rollover-count?
  & rollover-period?
  & rollover-size?
  & timestamp?
  & use-parent-handlers?
}
```

24.5.28 <logger>

child of: resin, cluster, host, web-app

<log> configures JDK 1.4 java.util.logger Logger level.

The log configuration describes log in detail.

ATTRIBUTE	DESCRIPTION	DEFAULT
level	the java.util.logging level: finest, finer, fine, config, info, warning, severe	info
name	the java.util.logging name, typically a classname	required
use-parent-handlers	if true, parent handlers are also invoked	true

<logger> Attributes

```
element logger {
  name
  & level?
  & use-parent-handlers?
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
  <log name="" level="all" path="log/debug.log"/>
  <logger name="com.caucho.java" level="fine"/>

  <cluster id="app-tier">
    ...
  </cluster>
</resin>
```

Example: compilation logging

24.5.29 <mail>

child of: resin, cluster, host, web-app

<mail> configures a javax.mail.Session object and makes it available in Resin-IoC/WebBeans. Mail properties can be configured using the properties attribute. Some of the most common properties can be configured directly on the <mail> tag.

ATTRIBUTE	DESCRIPTION	DEFAULT

authenticator	sets the javamail authenticator
debug	sets the mail.debug flag
from	sets the mail.from property
host	sets the mail.host property
imap-host	sets the mail.imap.host property
imap-port	sets the mail.imap.port property
imap-user	sets the mail.imap.user property
init	IoC configuration for other properties
jndi-name	JNDI name to store the mail Session
name	Resin-IoC/WebBeans @Named value
pop3-host	sets the mail.pop3.host property
pop3-port	sets the mail.pop3.port property
pop3-user	sets the mail.pop3.user property
properties	general mail properties in property file format
smtp-host	sets the mail.smtp.host property
smtp-port	sets the mail.smtp.port property
smtp-user	sets the mail.smtp.user property
store-protocol	sets the mail.store.protocol property
transport-protocol	sets the mail.transport.protocol property
user	sets the mail.user property

<mail> Attributes

```
element mail {
  authenticator?
  & debug?
  & from?
  & host?
  & imap-host?
  & imap-port?
  & imap-user?
  & init?
  & jndi-name?
  & name?
  & pop3-host?
  & pop3-port?
  & pop3-user?
  & smtp-host?
  & smtp-port?
  & smtp-user?
  & store-protocol?
  & transport-protocol?
  & user?
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

  <mail jndi-name="java:comp/env/mail">
    <from>noreply@foo.com</from>
    <smtp-host>localhost</smtp-host>
    <smtp-port>25</smtp-port>

    <properties>
      mail.smtp.starttls.enable=true
    </properties>
  </mail>
</web-app>
```

Example: mail

24.5.30 <reference>

child of: resin, cluster, host, web-app

<reference> configures a JNDI ObjectFactory. Some legacy resources are configured using an ObjectFactory syntax. The <reference> tag provides a compatible way to configure those objects. More modern resources should use <bean> or <component> for IoC configuration.

JNDI ObjectFactories are used to create objects from JNDI references. The <reference> tag configures the ObjectFactory and stores it in JNDI.

ATTRIBUTE	DESCRIPTION	DEFAULT
jndi-name	JNDI name for the reference. Since Resin 3.0	required

factory	Class name of the ObjectFactory. Resin 3.0	required
init	Bean-style initialization for the factory	none

<reference> Attributes

```
element reference {
  factory
  & jndi-name
  & init-param*
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

<reference>
  <jndi-name>hessian/hello</jndi-name>
  <factory>com.caucho.hessian.client.HessianProxyFactory</factory>
  <init url="http://localhost:8080/ejb/hello"/>
        type="test.HelloHome"/>
</reference>

</web-app>
```

Example: Hessian client reference

24.5.31 <remote-client>

child of: cluster, host, web-app

<remote-client> configures a proxy to a web-service. It uses a Java interface and a URI to select the web-service.

The URI is defined as: `protocol:url=location` , where location is typically a HTTP URL.

- See Resin remoting for more information, including how to write an adapter for Resin remoting.
- See the hello world tutorial for an example.

ATTRIBUTE	DESCRIPTION	DEFAULT
class	Class name of the protocol implementation	required (or `uri`)
init	IoC initialization for the protocol implementation	
name	@Named binding for Resin-IoC	

jndi-name	JNDI binding name
uri	Shortcut alias name for the protocol class

<remote-client> Attributes

URI	Description
`cxf` :url=http://foo.com/hello/cxf	Defines a cxf service. See http://wiki.caucho.com/CXF for more information.
`burlap` :url=http://foo.com/hello/burlap	Defines a burlap service at http://foo.com/hello/burlap
`hessian` :url=http://foo.com/hello/hessian	Defines a hessian service at http://foo.com/hello/hessian
`xfire` :url=http://foo.com/hello/cxf	Defines a xfire client. See http://wiki.caucho.com/XFire for more information.

remote-client protocols

```
element remote-client {
  (class|uri)
  & name?
  & jndi-name?
  & interface
}
```

24.5.32 <resin:choose>

resin:choose implements an if, elsif, else.

ATTRIBUTE	DESCRIPTION
resin:when	A configuration section executed when matching a test condition
resin:otherwise	A fallback section executed when the tests fail

<resin:choose> Attributes

The <resin:choose> schema is context-dependent. A <resin:choose> in a <web-app> will have <web-app> content, while a <resin:choose> in a <host> will have <host> content.

```
element resin:choose {
  resin:when*,
  resin:otherwise
}

element resin:when {
  attribute test { string },

   \texttt{context-dependent content}
}

element resin:otherwise {
   \texttt{context-dependent content}
}
```

```
<resin:choose>
  <resin:when test="${expr1}">
    ...
  </resin:when>

  <resin:when test="${expr2}">
    ...
  </resin:when>

  <resin:otherwise>
    ...
  </resin:otherwise>
<resin:choose>
```

Example: resin:choose usage pattern

<resin:when>

child of: resin:choose

<resin:when> conditionally configures a block within a <resin:choose> block. If the `test` matches, Resin will use the enclosed configuration.

ATTRIBUTE	DESCRIPTION
test	the test to perform

<resin:when> Attributes

```
element resin:when {
  attribute test { string },

   \texttt{context-dependent content}
}
```

<**resin:otherwise**>

child of: resin:choose

<resin:otherwise> is the catch-all configuration for a <resin:choose> block when none of the <resin:when> items match.

```
element resin:otherwise {
   \texttt{context-dependent content}
}
```

24.5.33 <resin:if>

resin:if executes part of the configuration file conditionally. resin:if can be particularly useful in combination with Java command-line properties like -Dfoo=bar to enable development mode or testing configuration.

ATTRIBUTE	DESCRIPTION	DEFAULT
test	the test to perform	required

<resin:if> Attributes

The resin:if schema is context-dependent. For example, resin:if in a <web-app> will have web-app content while resin:if in a <host> will have host content.

```
element resin:if {
  attribute test { string }

   \texttt{context-dependent content}
}
```

```
<resin xmlns="http://caucho.com/ns/resin"
        xmlns:core="http://caucho.com/ns/resin/core">

  <resin:if test="${system['development']}">
    <logger name="com.foo" level="finer"/>
  </resin:if>

  ...
</resin>
```

Example: enable debugging for -Ddevelopment

24.5.34 <resin:import>

<resin:import> reads configuration from another file or set of files. For example, the WEB-INF/web.xml and WEB-INF/resin-web.xml files are implemented as <resin:import> in the app-default.xml.

The target file is validated by the schema of the including context. So a resin:import in <web-app-default> will have a target with a top-level of <web-app>, and a resin:import in <cluster> will have a top-level tag of <cluster>.

ATTRIBUTE	DESCRIPTION	DEFAULT
path	a path to a file	either path or fileset is required
fileset	a fileset describing all the files to import.	either path or fileset is required
optional	if true, no error when file does not exist	false

<resin:import> Attributes

```
element import {
  (path | fileset)
  & optional?
}

element fileset {
  dir
  & exclude*
  & include*
}
```

The following example shows how Resin implements the WEB-INF/web.xml and WEB-INF/resin-web.xml files. Both are simply resin:import in a web-app-default. When Resin configures the web-app, it will process the web-app-default program, and call resin:import for the web.xml file.

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">
  <cluster id="app-tier">

    <web-app-default>
      <resin:import path="WEB-INF/web.xml" optional="true"/>
      <resin:import path="WEB-INF/resin-web.xml" optional="true"/>
    </web-app-default>

  </cluster>
</resin>
```

Example: import implementation of WEB-INF/web.xml

Virtual hosts can use resin:import to add a custom host.xml file. The host.xml can use any <host> attribute, including <host-name> and <host-alias> to customize the virtual host configuration.

```
<resin xmlns="http://caucho.com/ns/resin"
       xmlns:resin="http://caucho.com/ns/resin/core">
  <cluster id="app-tier">

    <host-deploy path="/var/www/hosts">
      <host-default>
        <resin:import path="host.xml" optional="true"/>

        <web-app-deploy path="webapps"/>
      </host-default>
    </web-app-default>

  </cluster>
</resin>
```

Example: adding host.xml in host-deploy

Some applications may want to split their configuration into multiple files using the fileset. For example, a Resin-IoC application might want to define beans in WEB-INF/beans/*.xml and give the web-app flexibility in which bean files to create.

```
<web-app xmlns="http://caucho.com/ns/resin"
         xmlns:core="http://caucho.com/ns/resin/core">

  <resin:import>
    <fileset dir="WEB-INF/beans">
      <include>*.xml</include>
    </fileset>
  </resin:import>

</web-app>
```

Example: Bean IoC fileset in resin-web.xml

24.5.35 <resin:message>

Logs a message to the given log file. The content of the element is the message.

```
element resin:message {
  string
}
```

```
<web-app xmlns="http://caucho.com/ns/resin"
         xmlns:resin="http://caucho.com/ns/resin/core">

  <resin:message>Starting server ${server.name}</resin:message>

</web-app>
```

logging in resin-web.xml

24.5.36 <resin:set>

resin:set adds an EL variable to the current context.

ATTRIBUTE	DESCRIPTION	DEFAULT
name	name of the variable to set	required
value	value	required

<resin:set> Attributes

```
element set {
  name
  & value
  & default
  & attribute * { string }
}
```

```
<resin xmlns="http://caucho.com/ns/resin"
          xmlns:resin="http://caucho.com/ns/resin/core">
  <resin:set name="root" value="/var/www"/>

  <cluster id="app-tier">
    <root-directory>${root}</root-directory>

    ...
  </cluster>
</resin>
```

Example: resin:set in resin.xml

24.5.37 <resource>

child of: resin, cluster, host, web-app

<resource> is an obsolete synonym for <bean> to define custom singletons. Applications should use the <bean> syntax instead.

24.5.38 <resource-adapter>

child of: resin, cluster, host, web-app

<resource-adapter> configures a JCA `ResourceAdapter` in combination with <connection-factory> for connections or <activation-spec for message listeners. See Resin messaging for typical uses.

`ResourceAdapters` can be deployed in .rar files, but this is not required by Resin. Instead, you can configure the ResourceAdapter directly.

A symbolic URI can be used in place of the ResourceAdapter's class name.

ATTRIBUTE	DESCRIPTION	DEFAULT
class	The classname of the ResourceAdapter implementation class	required (or `uri`)
init	IoC initialization for the ResourceAdapter	
jndi-name	JNDI name for binding the ResourceAdapter	
name	@Named binding for Resin-IoC injection.	
uri	Alias schema for the ResourceAdapter class name	

<resource-adapter> Attributes

```
element resource-adapter {
  (class | uri)
  & init?
  & name?
  & jndi-name?
}
```

24.5.39 <resource-deploy>

child of: resin, cluster, host-default, host, web-app-default, web-app

<resource-deploy> defines a deployment directory for .rar files.

Connectors and resources defined in .rar files must be deployed before they can be configured by connector. The <resource-deploy> tag specifies the directory for that deployment.

ATTRIBUTE	DESCRIPTION	DEFAULT
resource-deploy	Configures .rar deployment	required
path	Configures the path where users will place .rar files	required
expand-path	Configures the directory where Resin will expand rar files	the path value

<resource-deploy> Attributes

```
element resource-deploy {
  path
  & expand-directory?
  & expand-path?
  & resource-default?
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">

    <host id="">
      <resource-deploy path="deploy"/>
    </host>
  </cluster>
</resin>
```

Example: resource-deploy

24.5.40 <resource-ref>

child of: resin, cluster, host, web-app

<resource-ref> declares that the application needs a resouce configuration.

resource-ref is not directly used by Resin. It's a servlet configuration item intended to tell GUI tools which resources need configuration. Resource configuration in Resin uses the resource, reference, database, and ejb-server tags.

For backwards compatibility, Resin 2.1-style configuration files may still use resource-ref to configure resources, but it's recommended to convert the configuration.

```
element resource-ref {
  attribute id?,
  description*,
  res-ref-name,
  ref-type,
  res-auth,
  res-sharing-scope?
}
```

24.5.41 <scheduled-task>

<scheduled-task> schedules a job to be executed at specific times or after specific delays. The times can be specified by a cron syntax or by a simple delay parameter. The job can be either a `Runnable` bean, a method specified by an EL expression, or a URL.

When specified as an IoC bean, the bean task has full IoC capabilities, including injection, @TransactionAttribute aspects, interception and @Observes.

ATTRIBUTE	DESCRIPTION
class	the classname of the singleton bean to create
cron	a cron-style scheduling description
delay	a simple delay-based execution
init	IoC initialization for the bean
mbean-name	optional MBean name for JMX registration
method	EL expression for a method to be invoked as the task
name	optional IoC name for registering the task
period	how often the task should be invoked in simple mode
task	alternate task assignment for predefined beans

<scheduled-task> Attributes

```
element scheduled-task {
  class?
  & cron?
  & delay?
  & init?
  & mbean-name?
  & method?
  & name?
  & period?
  & task?
}
```

bean-style job configuration

The most common and flexible job configuration uses standard IoC bean-style configuration. The bean must implement `Runnable` . Like the <bean> tag, the `class` attribute specifies the `Runnable` class, and any `init` section configures the bean using Resin IoC configuration.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <scheduled-task class="qa.MyTask">
    <cron>*/5</cron>
  </scheduled-task>

</web-app>
```

Example: 5min cron bean task

task reference job configuration

The task bean can also be passed to the <scheduled-task> using a Resin-IoC EL reference. The name of the task bean would be defined previously, either in a <bean> or <component> or picked up by classpath scanning. Like the bean-style job configuration, the reference bean must implement `Runnable` .

```
<web-app xmlns="http://caucho.com/ns/resin">

  <scheduled-task task="#{taskBean}">
    <cron>0 0 *</cron>
  </scheduled-task>

</web-app>
```

Example: midnight cron bean task

method reference job configuration

<scheduled-task> can execute a method on a defined bean as the scheduler's task. The method is specified using EL reference syntax. At each trigger time, <scheduled-task> will invoke the EL method expression.

In the following example, the task invokes `myMethod()` on the `myBean` singleton every 1 hour.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <bean name="myBean" class="qa.MyBean"/>

  <scheduled-task method="#{myBean.myMethod}">
    <delay>10m</delay>
    <period>1h</period>
  </scheduled-task>

</web-app>
```

Example: 1h period method task

url job configuration

In a <web-app>, the <scheduled-task> can invoke a servlet URL at the trigger times. The task uses the servlet `RequestDispatcher` and forwards to the specified URL. The URL is relative to the <web-app> which contains the <scheduled-task.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <scheduled-task url="/cron.php">
    <cron>0 15 * * 0</cron>
  </scheduled-task>

</web-app>
```

Example: sunday cron url task

24.5.42 <servlet-hack>

child of: class-loader

Use of servlet-hack is discouraged. Using servlet-hack violates the JDK's classloader delegation model and can produce surprising ClassCastExceptions.

servlet-hack reverses the normal class loader order. Instead of parent classloaders having priority, child classloaders have priority.

```
element servlet-hack {
  boolean
}
```

24.5.43 <simple-loader>

child of: class-loader

<simple-loader> Configures a simple `WEB-INF/classes` -style class loader. .class files in the specified directory will be loaded without any special compilation steps (in contrast with compiling-loader.)

ATTRIBUTE	DESCRIPTION	DEFAULT
path	Filesystem path for the class loader. Since Resin 3.0	required
prefix	Class package prefix to only load to a subset of classes. Resin 3.0	none

<simple-loader> attributes

```
element simple-loader {
  path
  & prefix?
}
```

24.5.44 <stderr-log>

child of: resin, cluster, host-default, host, web-app-default, web-app

Configures the destination for `System.err` .

The log configuration describes stderr-log in detail.

ATTRIBUTE	DESCRIPTION	DEFAULT
archive-format	defines a format string for log rollover	
path	sets the VFS path for the log file	
path-format	sets a pattern for creating the VFS path for the messages	
rollover-count	sets the maximum number of rollover files	
rollover-period	sets the number of days before a log file rollover	1m
rollover-size	sets the maximum log size before a rollover	1g
timestamp	sets the formatting string for the timestamp label	

<stderr-log> Attributes

```
element stderr-log {
  (path | path-format)
  & archive-format?
  & rollover-period?
  & rollover-size?
  & rollover-count?
  & timestamp?
}
```

24.5.45 <stdout-log>

child of: resin, cluster, host-default, host, web-app-default, web-app

Configures the destination for `System.out` .

The log configuration describes stderr-log in detail.

ATTRIBUTE	DESCRIPTION	DEFAULT
archive-format	defines a format string for log rollover	
path	sets the VFS path for the log file	
path-format	sets a pattern for creating the VFS path for the messages	
rollover-count	sets the maximum number of rollover files	
rollover-period	sets the number of days before a log file rollover	1m
rollover-size	sets the maximum log size before a rollover	1g
timestamp	sets the formatting string for the timestamp label	

<stdout-log> Attributes

```
element stdout-log {
  (path | path-format)
  & archive-format?
  & rollover-period?
  & rollover-size?
  & rollover-count?
  & timestamp?
}
```

24.5.46 <system-property>

child of: resin, cluster, host, web-app

Sets a Java system property. The effect is the same as if you had called before starting Resin.

```
element system-property {
  attribute * { string }+
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
  <system-property foo="bar"/>
</resin>
```

Example: setting system property

24.5.47 <temp-dir>

child of: resin, cluster, host-default, host, web-app-default, web-app**default:** Defaults to `WEB-INF/tmp`

<temp-dir> configures the application temp directory. This is the path used in `javax.servlet.context.tempdir` .

```
element temp-dir {
  string
}
```

24.5.48 <tree-loader>

child of: class-loader

<tree-loader> configures a jar library, `WEB-INF/lib` -style class loader similar to , but will also find `.jar` and `.zip` files in subdirectories.

ATTRIBUTE	DESCRIPTION	DEFAULT
path	Filesystem path for the class loader. Since Resin 3.0	required

<tree-loader> Attributes

```
element tree-loader {
  path
}
```

24.5.49 <work-dir>

child of: resin, config, host, web-app**default:** Defaults to `WEB-INF/work`

<work-dir> configures a work directory for automatically generated code, e.g. for JSP, PHP, and JPA classes.

```
element work-dir {
  string
}
```

24.6 URL rewrite tags

24.6.1 <and>

child of: when, unless

Contains one or more conditions and evaluates to true if all of the contained conditions evaluate to true, false if any of the contained conditions does not evaluate to true.

24.6.2 auth-type

child of: when, unless

Evaluates to true if the authorization mechanism used in the request is the given value. The comparison is always case insensitive.

If the auth-type is not "none", or if the auth-type of the request is not "none", a `Cache-Control` header containing " `private` " is added to the response.

auth-type	BASIC, CLIENT-CERT, DIGEST, FORM, or NONE	required
send-vary	Send a `Vary` header containing " `Cookie` " as part of the response	true

24.6.3 cookie

child of: when, unless

Evaluates to true if the value of a cookie with a given `name` matches a regular expression `regexp` , false if it does not or if the cookie does not exist.

The `Vary` header of the response is updated to include "Cookie", which indicates to the browser and any intervening proxy cache that the response varies based on the submitted value of cookies.

cookie	Name of the cookie	required
regexp	The regular expression to match	optional
send-vary	Send a `Vary` header containing " `Cookie` " as part of the response	true

24.6.4 disable-at

Cron syntax for specifying a time or times that that the rule should be disabled. In conjunction, `enable-at` and `disable-at` provide a means for scheduling the enablement of a rule.

24.6.5 <dispatch>

child of: rewrite-dispatch, match

If <dispatch> matches the current URL, the rest of the items in the enclosing rewrite-dispatch are not considered and the request passes immediately to normal servlet evaluation. <dispatch> is often used to specify URLs which should be handled normally before a more general pattern which modifies the URL.

For example, the following pattern uses <dispatch> to handle images and *.php files normally, but forward's all other requests to /index.php.

```
<web-app xmlns="http://caucho.com/ns/resin">

 <rewrite-dispatch>
   <dispatch regexp="\.(php|gif|css|jpg|png)"/>
   <forward regexp="^" target="/index.php"/>
 </rewrite-dispatch>

</web-app>
```

Mediawiki dispatch in resin-web.xml

disable-at	A cron syntax time specification for triggering disablement of the rule	none
enable-at	A cron syntax time specification for triggering enablement of the rule	none
enabled	False to start with the rule initially disabled	true
name	A name to use for registering a JMX mbean with type=RewriteRule	do not register an mbean
regexp	A regexp that must match the URL	optional
when	A condition which must evaluate to true	optional
unless	A condition which must not evaluate to true	optional

24.6.6 dispatch-type

The dispatch-type lets you configure rewrite-dispatch for REQUEST, FORWARD, or INCLUDE. By default, the rewrite-dispatch is REQUEST.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <rewrite-dispatch>
    <dispatch-type>FORWARD</dispatch-type>

    <forward regexp="/foo" target="/bar.jsp"/>
  </rewrite-dispatch>

</web-app>
```

Example: FORWARD dispatch

24.6.7 enable-at

Cron syntax for specifying a time or times that that the rule should be enabled, see <disable-at/>.

24.6.8 enabled

Set's the initial state of the rule, default is true. In conjunction with name, an initial enabled value of "false" is valuable for rules which are to be turned on and turned off at runtime.

24.6.9 <forbidden>

child of: rewrite-dispatch, match

<forbidden> sents a 403 forbidden message to the browser for a matching URL.

```
<web-app xmlns="http://caucho.com/ns/resin">

 <rewrite-dispatch>
   <forbidden regexp="^/protected"/>
 </rewrite-dispatch>

</web-app>
```

Forbidding a directory

disable-at	A cron syntax time specification for triggering disablement of the rule	none
enable-at	A cron syntax time specification for triggering enablement of the rule	none
enabled	False to start with the rule initially disabled	true
name	A name to use for registering a JMX mbean with type=RewriteRule	do not register an mbean
regexp	A regexp that must match the URL	optional
when	A condition which must evaluate to true	optional
unless	A condition which must not evaluate to true	optional

24.6.10 <forward>

child of: rewrite-dispatch, match

<forward> rewrites the current URL, forwarding it to the target using the servlet forward() call. Because <forward> is internal, it will have better performance than a <redirect> when the target URL is on the same server, the browser will not know that the underlying resource has moved.

```
<web-app xmlns="http://caucho.com/ns/resin">

 <rewrite-dispatch>
   <forward regexp="^/old" target="/new"/>
 </rewrite-dispatch>

</web-app>
```

Forwarding to a new URL

disable-at	A cron syntax time specification for triggering disablement of the rule	none

enable-at	A cron syntax time specification for triggering enablement of the rule	none
enabled	False to start with the rule initially disabled	true
name	A name to use for registering a JMX mbean with type=RewriteRule	do not register an mbean
regexp	A regexp that must match the URL	optional
target	The target of the the foreward	required
when	A condition which must evaluate to true	optional
unless	A condition which must not evaluate to true	optional

24.6.11 header

child of: when, unless

Evaluates to true if the value of a header with a given `name` matches a regular expression `regexp` , false if it does not or if the header does not exist.

The `Vary` header of the response is updated to include the header name which indicates to the browser and any intervening proxy cache that the response varies based on the submitted value of the header.

header	Name of the header	required
regexp	The regular expression to match	optional
send-vary	Send a `Vary` header containing the header name as part of the response	true

24.6.12 <gone>

child of: rewrite-dispatch, match

<gone> sents a 410 gone response to the browser for a matching URL.

```
<web-app xmlns="http://caucho.com/ns/resin">

 <rewrite-dispatch>
   <gone regexp="^/protected"/>
 </rewrite-dispatch>

</web-app>
```

Hiding a directory

disable-at	A cron syntax time specification for triggering disablement of the rule	none
enable-at	A cron syntax time specification for triggering enablement of the rule	none
enabled	False to start with the rule initially disabled	true
name	A name to use for registering a JMX mbean with type=RewriteRule	do not register an mbean
regexp	A regexp that must match the URL	optional
when	A condition which must evaluate to true	optional
unless	A condition which must not evaluate to true	optional

24.6.13 <import>

child of: rewrite-dispatch, match

<import> loads rules from an external xml file.

```
<web-app xmlns="http://caucho.com/ns/resin"
         xmlns:resin="http://caucho.com/ns/resin/core">

  <rewrite-dispatch>
    <import path='/WEB-INF/rewrite-maintenance.xml' enabled="false"/>
    <import path='/WEB-INF/rewrite-rules.xml'/>
  </rewrite-dispatch>

</web-app>
```

Importing rules from an external file

```
<rewrite-dispatch xmlns="http://caucho.com/ns/resin"
                  xmlns:resin="http://caucho.com/ns/resin/core">

   <dispatch regexp="\.(css|gif|jpg|pdf|png)"/>

   <forward regexp=".*" target="/maintenance.html"/>

</rewrite-dispatch>
```

An external file containing rules

Changes to the external file are checked for at a frequency determined by `dependency-check-interval` or when the `update()` operation of the mbean is invoked. If an error occurs when loading the modified file, the contents of the old file are used until the errors are corrected or the server is restarted. The log file will contain details for the error, and the `RedeployError` property of the mbean wil contain a description of the error.

Each <import> registers an mbean of type RewriteImport, with a full a name like "resin:Host=default,WebApp=/,name=Foo,type=RewriteRule" and an interface of . It provides the `update()` , `start()` and `stop()` operations. The `update()` operations forces an immediate check for modifications to the external file.

disable-at	A cron syntax time specification for triggering disablement of the rule	none
dependency-check-interval	The frequency with which to check the external file for changes	The dependency-check-interval of the environment
enable-at	A cron syntax time specification for triggering enablement of the rule	none

enabled	False to start with the rule initially disabled	true
name	A name to use for registering a JMX mbean with type=RewriteImport	the value of `path`
optional	If false, do not require that the path exists	true
path	A path to an external xml file	required

24.6.14 <load-balance>

child of: rewrite-dispatch, match

<load-balance> forwards requests from a web-tier server to a cluster of app-tier servers for load-balancing. Load balancing provides scalability by splitting load among many application servers and increases reliability by avoiding servers which are upgrading or restarting.

In the following example, the web-tier load-balances all traffic to a cluster of backend application servers. Because the web-tier has a proxy cache, static pages and cached pages will be served directory from the web-tier.

<load-balance> requires Resin Professional.

```
<resin xmlns="http://caucho.com/ns/resin">

  <cluster id="app-tier">
    <server id="app-a" address="192.168.3.1">
    <server id="app-b" address="192.168.3.2">
    ...
  </cluster>

  <cluster id="web-tier">
    <server id="web-a" address="192.168.2.1">
      <http port="80"/>
    </server>

    <cache/>

    <host id="">
      <web-app id="/">

        <rewrite-dispatch>
          <load-balance regexp="" cluster="app-tier"/>
        </rewrite-dispatch>

      </web-app>
    </host>

</resin>
```

Load balancing to an app-tier

cluster	The cluster that gets the matching requests	required
disable-at	A cron syntax time specification for triggering disablement of the rule	none
enable-at	A cron syntax time specification for triggering enablement of the rule	none
enabled	False to start with the rule initially disabled	true
name	A name to use for registering a JMX mbean with type=RewriteRule	do not register an mbean
regexp	A regexp that must match the URL	optional
sticky-sessions	Whether or not sessions should be sticky	true

strategy	The load balancing strategy, 'round-robin' or 'least-connection'	least-connection
when	A condition which must evaluate to true	optional
unless	A condition which must not evaluate to true	optional

24.6.15 locale

child of: when, unless

Evaluates to true if the value of the Locale matches a regular expression, false if it does not. The comparison is always case insensitive.

The Locale is a normalization of the value of the Accept-Language header. For example "fr" remains "fr" and "FR-CA" becomes "fr_CA". If the request does not include the Accept-Language header then the Locale is the default Locale for the server.

```
<web-app xmlns="http://caucho.com/ns/resin"
         xmlns:resin="http://caucho.com/ns/resin/core">

  <rewrite-dispatch>
    <forward regexp='^/' target='/fr/'>
      <when locale="^fr"/>
    </forward>

    <forward regexp='^/' target='/en/'/>
  </rewrite-dispatch>
</web-app>
```

locale	Regular expression to match	required
send-vary	Send a `Vary` header containing " `Accept-Language` " as part of the response	true

24.6.16 local-port

child of: when, unless

Evaluates to true if the value of the request's local port is the specified value,

false if it does not. The local port is the port that Resin has bound to and has used to receive the request.

local-port	The local port to match.	required

24.6.17 method

child of: when, unless

Evaluates to true if the http method used in the request is the given value. The comparison is always case insensitive. Common methods include HEAD, GET, POST, DELETE, OPTIONS, PUT, and TRACE, although a method by any name may be used by a client.

method	The method to match.	required

24.6.18 <match>

child of: rewrite-dispatch, match

<match> provides a way to group more specific rules based on the url. The contents of the <match> are any rule, and they are considered only if the regexp of the <match> is satsified and the match is `enabled` .

Judicious use of <match> will provide better performance in situations where there are many rules and some of them contain when or if conditions.

<match> also provides a convenient way to group a set of rules that can be enabled or disabled using an mbean. If the `name` attribute is specified for the <match> then an mbean with a name like `resin:Host=default,WebApp=/,name=`*`name`*`,type=RewriteRule` is registered and the `start()` and `stop()` operations are available.

The documentation for <rewrite-dispatch> provides a table of all rules. In addition to the rules that can be contained within <match>, the following configuration tags are available.

disable-at	A cron syntax time specification for triggering disablement of the rule	none
enable-at	A cron syntax time specification for triggering enablement of the rule	none

enabled	False to start with the rule initially disabled	true
name	A name to use for registering a JMX mbean with type=RewriteRule	do not register an mbean
regexp	A regexp that must match the URL	optional
when	A condition which must evaluate to true	optional
unless	A condition which must not evaluate to true	optional

24.6.19 <moved-permanently>

child of: rewrite-dispatch, match

<moved-permanently> sends a HTTP 301 moved permanently response to the browser, indicating that the resource has moved.

The following example causes all requests to the "old.com" host to be immediately redirected to "new.com". The urls will be maintained, so for example a request to `http://old.com/hello/world.html?foo=bar` will get redirected to `http://new.com/hello/world.html?foo=bar` .

```
<host id="old.com">
  <web-app id="/">
    <rewrite-dispatch>
      <moved-permanently regexp="^/(.*)$"
                         target="http://new.com/$1"/>
    </rewrite-dispatch>
  </web-app>
</host>
```

Forwarding to a new virtual host

disable-at	A cron syntax time specification for triggering disablement of the rule	none
enable-at	A cron syntax time specification for triggering enablement of the rule	none
enabled	False to start with the rule initially disabled	true

name	A name to use for registering a JMX mbean with type=RewriteRule	do not register an mbean
regexp	A regexp that must match the URL	optional
when	A condition which must evaluate to true	optional
unless	A condition which must not evaluate to true	optional

24.6.20 name

A rule can be given a *name*, which causes a JMX mbean to be registered. The mbean has a name like "resin:Host=default,WebApp=/,name=Foo,type=RewriteRule" and an interface of . It provides the `start()` and `stop()` operations.

24.6.21 <not>

child of: when, unless

Contains one or more conditions and evaluates to true only if none of the contained conditions evaluate to true, false if any of the contained conditions evaluate to true.

24.6.22 <or>

child of: when, unless

Contains one or more conditions and evaluates to true if any of the contained conditions evaluate to true, false only if all of the contained conditions evaluate to false.

24.6.23 <not-found>

child of: rewrite-dispatch, match

<not-found> sents a 404 not found to the browser for a matching URL.

```
<web-app xmlns="http://caucho.com/ns/resin">

 <rewrite-dispatch>
   <not-found regexp="^/protected"/>
 </rewrite-dispatch>

</web-app>
```

Hiding a directory

disable-at	A cron syntax time specification for triggering disablement of the rule	none
enable-at	A cron syntax time specification for triggering enablement of the rule	none
enabled	False to start with the rule initially disabled	true
name	A name to use for registering a JMX mbean with type=RewriteRule	do not register an mbean
regexp	A regexp that must match the URL	optional
when	A condition which must evaluate to true	optional
unless	A condition which must not evaluate to true	optional

24.6.24 query-param

child of: when, unless

Evaluates to true if the value of a query parameter with a given `name` matches a regular expression `regexp` , false if it does not or if the parameter does not exist.

Form parameters submitted in the body of a POST are not available for the comparison performed by query-param.

query-param	Name of the parameter	required
regexp	The regular expression to match	optional

24.6.25 <redirect>

child of: rewrite-dispatch, match

<redirect> sends a HTTP 302 redirect response to the browser, indicating that the resource has moved.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <rewrite-dispatch>
    <redirect regexp="^/(foo)" target="http://$1.bar.com"/>
  </rewrite-dispatch>

</web-app>
```

Forwarding to a new virtual host

```
  <host ... >
     <rewrite-dispatch>
        <redirect regexp="^" target="https://${host.name}">
          <when secure="false"/>
        </redirect>
      </rewrite-dispatch>
      ...
   </host>
```

Redirecting all http:// requests to https://

disable-at	A cron syntax time specification for triggering disablement of the rule	none
enable-at	A cron syntax time specification for triggering enablement of the rule	none
enabled	False to start with the rule initially disabled	true
name	A name to use for registering a JMX mbean with type=RewriteRule	do not register an mbean
regexp	A regexp that must match the URL	optional
when	A condition which must evaluate to true	optional
unless	A condition which must not evaluate to true	optional

24.6.26 <regexp>

The regular expression in a regexp tag or attribute is a standard Java regular expression.

To make the regexp case insensitive, you can use the `(?i:...)` syntax:

```
<web-app xmlns="http://caucho.com/ns/resin"
         xmlns:resin="http://caucho.com/ns/resin/core">
  <rewrite-dispatch>
    <forbidden regexp='(?i:^/foo)'/>
  </rewrite-dispatch>
</web-app>
```

case-insensitive match

24.6.27 remote-addr

child of: when, unless

Evaluates to true if the remote address of the client matches the specified ip address.

A `Cache-Control` header containing " `private` " is added to the response.

```
<web-app xmlns="http://caucho.com/ns/resin"
         xmlns:resin="http://caucho.com/ns/resin/core">
  <rewrite-dispatch>
    <forbidden regexp='^/foo'>
      <unless remote-addr="192.168.2.1/24"/>
    </forbidden>

  </rewrite-dispatch>
</web-app>
```

Forbidding admin access to clients outside of the local network

remote-addr	An IP address, optionally followed by a `/` and a number of bits to match.	required

24.6.28 remote-user

child of: when, unless

Evaluates to true if the user is authenticated and has the specified name.

If there is a current user, a `Cache-Control` header containing " `private` " is added to the response.

remote-user	A user name.	required
send-vary	Send a `Vary` header containing " `Cookie` " as part of the response	true

24.6.29 <rewrite>

child of: rewrite-dispatch, match

<rewrite> rewrites the current URL, continuing processing of the <rewrite-dispatch> tags. <rewrite> can be used as an intermediate rewriting stage for more complicated patterns.

24.6.30 <rewrite-real-path>

child of: web-app

<rewrite-real-path> configures an alias for the getRealPath() call, i.e. an enhancement to the <path-mapping> tag. It provides a virtual directory mapping that allows url patterns in the web application to have a local path anywhere on the physical filesystem.

The source path is a URL, the target path is a real filesystem path.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <rewrite-real-path>
    <real-path regexp="^/images" target="/usr/local/share/images"/>
  </rewrite-real-path>
```

`/images located outside of /var/www`

```
<web-app xmlns="http://caucho.com/ns/resin">

  <rewrite-real-path>
    <real-path regexp="^/" target="WEB-INF/SugarCE-Full-5.0.0a"/>
  </rewrite-real-path>
```

`Managing versions of software bundles`

Merge paths

A merge path is a collection of paths, similar to a classpath. It allows the specification of multiple paths that are merged together and presented to the application as if it was one unified path. Merge paths are specified with a syntax of " `merge:(path1[;path2[;pathN]])` ". If a file is in `path1` it overrides a file with the same name in `path2` , if a file is not in `path1` then the file will be matched to `path2` .

Merge paths are useful for specifying customization directories that allow for the insertion of files that replace the usual file that is shipped with a web application.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <rewrite-real-path>
    <real-path regexp="^/images" target="merge:(custom-images;images)"/>
  </rewrite-real-path>
```

Merge path: replacing some images with custom images

```
<web-app xmlns="http://caucho.com/ns/resin">

  <rewrite-real-path>
    <real-path regexp="^/" target="merge:(WEB-INF/SugarCE-local;WEB-INF/SugarCE-Full-5.0.0a)"/
  </rewrite-real-path>
```

Merge path: managing the versioning and customization of a php application

24.6.31 <rewrite-dispatch>

child of: server, host, web-app

<rewrite-dispatch> evaluates the child tags in order. The first matching tag dispatches the request. If no children match, the request uses the standard servlet handling.

The child tags rewrite and dispatch based on regexp patterns.

```
<web-app xmlns="http://caucho.com/ns/resin">

 <rewrite-dispatch>
   <dispatch regexp="\.(php|gif|css|jpg|png)"/>
   <forward regexp="^" target="/index.php"/>
 </rewrite-dispatch>

</web-app>
```

Mediawiki dispatch in resin-web.xml

<dispatch>	Dispatch the request immediately and do not process any more rules
<forbidden>	Send a 403 Forbidden response to browser and do not process any more rules

<forward>	Rewrite the URL and internally forward the request without processing any more rules
<forbidden>	Send a 410 Gone response to the browser and do not process any more rules
<import>	Import rules from an external file
<load-balance>	Forward requests to another instance or instances of Resin and do not process any more rules
<match>	Group more specific rules
<moved-permanently>	Send a 310 Moved Permanently response to the browser and do not process any more rules
<redirect>	Send a 302 Moved response to the browser and do not process any more rules
<rewrite>	Rewrite the current URL and continue processing rules
<set>	Set a property of the request or response and continue processing rules

24.6.32 secure

child of: when, unless

<secure> specifies a condition for the ssl status of the request. If `secure` is true, then the request must be using ssl for the rule to match. If `secure` is false, then the request must not be using ssl for the rule to match. If `secure` is not specified, the ssl status of the request is not considered when testing if the rule matches.

```
<web-app xmlns="http://caucho.com/ns/resin">
  <rewrite-dispatch>
    <redirect regexp='^/foo' target='/secure'>
      <when secure='true'/>
    </redirect>

    <redirect regexp='^/foo' target='/insecure'>
      <when secure='false'/>
    </redirect>

    <redirect regexp='^/bar' target='/public'/>
  </rewrite-dispatch>
</web-app>
```

secure example

```
http://localhost:8080/foo/index.html
  --> redirect to /insecure/index.html

https://localhost:8080/foo/index.html
  --> redirect to /secure/index.html

http://localhost:8080/bar/index.html
  --> redirect to /public/index.html

https://localhost:8080/bar/index.html
  --> redirect to /public/index.html
```

24.6.33 server-name

child of: when, unless

Evaluates to true if the value of the request's server name matches the specified regexp, false if it does not. The comparison is always case insensitive. The server name is the name that the client has used to connect to the server and may not be the name of the actual server that is hosting Resin.

Most potential problems that can be solved by `server-name` are more efficiently solved using virtual hosts and the host-alias tag.

server-name	A regular expression to match.	required

24.6.34 server-port

child of: when, unless

Evaluates to true if the value of the request's server port is the specified value, false if it does not. The server port is the port that the client has used to connect to the server and may not be the actual port that Resin is bound to.

server-port	The server port to match.	required

24.6.35 <set>

child of: rewrite-dispatch, match

<set> is used to set properties of the request or response before continuing on to the next rule. The properties are maintained after the rewriting is complete, for example a request that becomes secure because of <set> will continue to be regarded as secure when the request eventually reaches a jsp or servlet.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <rewrite-dispatch>
    <set request-secure="true">
      <when header="X-SSL-cipher"/>
    </set>

    <forward regexp="^/checkout" target="https://hogwarts.com/checkout">
      <when secure="false"/>
    </forward>

  </rewrite-dispatch>

</web-app>
```

Making the request secure in response to a header from a hardware load balancer

disable-at	A cron syntax time specification for triggering disablement of the rule	none
enable-at	A cron syntax time specification for triggering enablement of the rule	none
enabled	False to start with the rule initially disabled	true
name	A name to use for registering a JMX mbean with type=RewriteRule	do not register an mbean

regexp	A regexp that must match the URL	optional
request-character-encoding	Sets the character encoding of the request, used for parsing POST data, request.setCharacterEncod	none
request-secure	Sets the value of request.isSecure()	none
response-character-encoding	Sets the value of the character encoding sent for the response with response.setCharacterEnco If a Content-Type is not set either with a set rule, or by some other process involved in servicing the request, this has no effect.	none
response-content-type	Sets the value of the content type sent for the response. If the final destination is a file with a corresponding mime-mapping, the mime-mapping will override the content-type set here	none
when	A condition which must evaluate to true	optional
unless	A condition which must not evaluate to true	optional

24.6.36 <unless>

Contains a condition that further refines the match of the enclosing tag beyond the information provided by the url. If the condition does not evaluate to true, then the enclosing tag is applied.

See <when> for a description of possible conditions.

24.6.37 user-in-role

child of: when, unless

Evaluates to true if the user is authenticated and in the specified role. The special role of '*' means any role.

If the current user is any role, a `Cache-Control` header containing " `private` " is added to the response.

user-in-role	A role name.	required
send-vary	Send a `Vary` header containing " `Cookie` " as part of the response	true

24.6.38 <when>

Contains a condition that further refines the match of the enclosing tag beyond the information provided by the url. If the condition evaluates to true, then the enclosing tag is applied.

<and>	A subgroup of conditions that must all pass	
auth-type	Evaluates to true if the authorization mechanism used in the request is the given value	
cookie	Evaluates to true if a cookie exists and matches an optional `regexp`	
header	Evaluates to true if a request header exists and matches an optional `regexp`	
locale	Evaluates to true if the request locale matches the specified value	
local-port	Evaluates to true if the local port is the specified value (the local port is the actual port that Resin is using)	
method	Evaluates to true if the http method used for the request is the specified value	

<not>	A subgroup of conditions, none of which may pass
<or>	A subgroup of conditions, any of which may pass
query-param	Evaluates to true if a query parameter exists and matches an optional `regexp`
regexp	Used in conjunction with `cookie` , `header` , and `query-param`
remote-addr	Evaluates to true if the remote address of the client matches the specified ip address
remote-user	Evaluates to true if the user is authenticated and has the specified name
secure	Evaluates to true if the ssl status of the request is equal to the specified value of true or false
server-name	Evaluates to true if the server name used by the client matches the specified value
server-port	Evaluates to true if the port used by the client equals the specified value
user-in-role	Evaluates to true if the user is authenticated and in the specified role

Logging and Debugging

Logging for the name `com.caucho.server.rewrite` at the "finer" level reveals successful matches. At the "finest" level both successful and unsuccessful matches are logged.

```
<logger name="com.caucho.server.rewrite" level="finest"/>
```

Logging example

```
[1998/05/08 02:51:31.000] forward ^/foo: '/baz/test.jsp'  no match
[1998/05/08 02:51:31.000] forward ^/bar: '/baz/test.jsp'  no match
[1998/05/08 02:51:31.000] forward ^/baz: '/baz/test.jsp'  -->  '/hogwarts/test.jsp'
```

Examples

Redirect http:// requests to https:// requests

The desired behaviour is to redirect plain connections to SSL connections.

```
http://anything.com/anything.html
  redirect => https://anything.com/anything.html
```

Desired behaviour

```
<host ...>
  ...
  <rewrite-dispatch>
    <redirect regexp="^" target="https://${host.name}">
      <when secure="false"/>
    </redirect>
  </rewrite-dispatch>
  ...
</host>
```

Configuration

Make an alias for a web-app

The desired behaviour is to make it so that a web-app will match more than one url pattern. For example, a web-app is deployed in `webapps/physics` and available at `http://hostname/physics/` , the desired behaviour is to allow a request to `http://hostname/classroom/physics` to end up at the `/physics` web-app.

```
http://hostname/classroom/physics
  forward => http://hostname/physics

http://hostname/classroom/physics/anything
  forward => http://hostname/physics/anything
```

Desired behaviour

The rewrite-dispatch tag is used at the <host> level. If it was placed in a <web-app> then it would be too late to forward to a different web-app because Resin would have already resolved the web-app.

```
<host>
  <rewrite-dispatch>
    <forward regexp="^/classroom/physics" target="/physics"/>
  </rewrite-dispatch>
   ...
```

Configuration

Forward based on host name

The desired behaviour is to internally forward requests based on the host name.

```
http://gryffindor.hogwarts.com/anything.html
  forward => /gryffindor/*

http://slytherin.hogwarts.com/anything.html
  forward => /slytherin/anything.html

http://hogwarts.com/anything.html
  forward => /anything.html
```

Desired behaviour

The rewrite-dispatch tag is used at the <cluster> level. If it was placed in the <host> or the <web-app> then it would be too late to forward to a different host because Resin would have already resolved the host.

```
<cluster>
  ...
  <rewrite-dispatch>
    <forward regexp="http://gryffindor\.[^/]+" target="/gryffindor/"/>
    <forward regexp="http://slytherin\.[^/]+" target="/slytherin/"/>
  </rewrite-dispatch>
  ...
</cluster>
```

Configuration

24.7 server: Server tag configuration

See Also

- See the index for a list of all the tags.
- See <cluster> tag configuration

24.7.1 <accept-listen-backlog>

child of: http, connection-port, protocol

<accept-listen-backlog> configures operating system TCP listen queue size for the port.

When a browser connects to a server, the server's operating system handles the TCP initialization before handing the socket to the server's application. The operating system will hold the opened connections in a small queue, until the application is ready to receive them. When the queue fills up, the operating system will start refusing new connections.**default:** 100

24.7.2 <accept-thread-max>

child of: http, connection-port, protocol

<accept-thread-max> configures the maximum number of threads listening for new connections on this port. <accept-thread-max> works with <accept-thread-min> to handle spiky loads without creating and destroying too many threads.

Socket connections are associated with a thread which handles the request. In Resin, a number of threads wait to accept a new connection and then handle the request. <accept-thread-max> specifies the maximum number of threads which are waiting for a new connection.

Larger values handle spiky loads better but require more threads to wait for the connections. Smaller values use less threads, but may be slower handling spikes.**default:** 10

24.7.3 <accept-thread-min>

child of: http, connection-port, protocol

<accept-thread-min> configures the minimum number of threads listening for new connections on this port <accept-thread-min> works with <accept-thread-max> to handle spiky loads without creating and destroying too many threads.

Socket connections are associated with a thread which handles the request. In Resin, a number of threads wait to accept a new connection and then handle the request. <accept-thread-min> specifies the minimum number of threads which are waiting for a new connection. If many connections appear rapidly with a small value of <accept-thread-min>, the application may pause until a new thread is available for the new connection.

Larger values handle spiky loads better but require more threads to wait for the connections. Smaller values use less threads, but may be slower handling spikes.

24.7.4 <address>

child of: server

The server <address> defines the IP interface for Resin cluster communication and load balancing. It will be an internal IP address like 192.168.* for a clustered configuration or 127.* for a single-server configuration. No wild cards are allowed because the other cluster servers and load balancer use the address to connect to the server.**default:** 127.0.0.1

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server-default>
      <http port="80"/>
    </server-default>

    <server id="web-a" address="192.168.1.1" port="6800"/>
    <server id="web-b" address="192.168.1.2" port="6800"/>

    ...
  </cluster>

  <cluster id="app-tier">
    <server id="app-a" address="192.168.2.11" port="6800"/>
    <server id="app-b" address="192.168.2.12" port="6800"/>

    ...
  </cluster>
</resin>
```

server address

24.7.5 <cluster-port>

child of: server

<cluster-port> configures the cluster and load balancing socket, for load balancing, distributed sessions, and distributed management.

When configuring Resin in a load-balanced cluster, each Resin instance will have its own <srun> configuration, which Resin uses for distributed session management and for the load balancing itself.

When configuring multiple JVMs, each <srun> has a unique <id> which allows the -server command-line to select which ports the server should listen to.

address	hostname of the interface to listen to	*
jsse-ssl	configures the port to use JSSE for SSL	none
openssl	configures the port to use OpenSSL	none
port	port to listen to	required
read-timeout	timeout waiting to read from idle client	65s

write-timeout	timeout waiting to write to idle client	65s
accept-listen-backlog	The socket factory's listen backlog for receiving sockets	100
tcp-no-delay	sets the NO_DELAY socket parameter	true

The class that corresponds to <srun> is

24.7.6 <group-name>

child of: server

<group-name> configures the operating system group Resin should run as. Since the HTTP port 80 is protected in Unix, the web server needs to start as root to bind to port 80. For security, Resin should switch to a non-root user after binding to port 80.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">

    <server-default>
      <http port="80"/>

      <user-name>resin</user-name>
      <group-name>www</group-name>
    </server-default>

    <server id="web-a"/>
    ...
  </cluster>
</resin>
```

resin.xml with user-name

24.7.7 <http>

child of: server

<http> configures a HTTP or HTTPS port listening for HTTP requests.

When configuring multiple JVMs, each <http> will have a unique <server-id> which allows the -server command-line to select which ports the server should listen to.

address	IP address of the interface to listen to	*
port	port to listen to	required

tcp-no-delay	sets the NO_DELAY socket parameter	true
read-timeout	timeout waiting to read from idle client	65s
write-timeout	timeout waiting to write to idle client	65s
socket-listen-backlog	The socket factory's listen backlog for receiving sockets	100
virtual-host	forces all requests to this <http> to use the named virtual host	none
openssl	configures the port to use OpenSSL	none
jsse-ssl	configures the port to use JSSE for SSL	none

The `virtual-host` attribute overrides the browser's Host directive, specifying the explicit host and port for `request.getServerName()` and `getServerPort()` . It is not used in most virtual host configurations. Only IP-based virtual hosts which wish to ignore the browser's Host will use @virtual-host.

24.7.8 <jvm-arg>

child of: server

<jvm-arg> configures JVM arguments to be passed to Resin on the command line, typically -X memory parameters and -D defines.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server-default>
      <jvm-arg>-Xmx512m</jvm-arg>
      <jvm-arg>-Xss1m</jvm-arg>
      <jvm-arg>-verbosegc</jvm-arg>
    </server-default>

    <server id="app-a" address="192.168.2.10"/>

    ...
  </cluster>
</resin>
```

standard jvm-args

24.7.9 <jvm-classpath>

child of: server

<jvm-classpath> adds a classpath entry when starting the JVM.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server-default>
      <jvm-classpath>/tmp/test-classpath;/jvm-classpath>
    </server-default>

    <server id="app-a" address="192.168.2.10"/>

    ...
  </cluster>
</resin>
```

adding a classpath

24.7.10 <keepalive-max>

child of: server

<keepalive-max> configures the maximum number of sockets which can be used directly for keepalive connections. In Resin Professional, the select manager allows for a much larger number of keepalive sockets, since it can detach threads from connections. Without the select manager, each connection is associated with a thread.

A value of -1 disables keepalives.

Keepalives are an important TCP technique used with HTTP and Resin's load-balancing to avoid the heavy network cost of creating a new socket. Since an initial HTTP request is usually immediately followed by extra requests to load files like images and stylesheets, it's generally more efficient to keep the socket open for a short time instead of creating a new one. The socket keepalive is even more important for Resin's load balancing, to avoid creating extra sockets between the web-tier and the app-tier and to make distributed sessions more efficient.

Higher values of <keepalive-max> improve network efficiency but increase the number of threads waiting for new client data.**default:** 100

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server-default>
      <http port="80"/>

      <thread-max>512</thread-max>

      <keepalive-max>100</keepalive-max>
    </server-default>

    <server id="web-a" address="192.168.0.10"/>
        ...
  </cluster>
</resin>
```

keepalive-thread-max in resin.xml

24.7.11 <keepalive-select-enable>

child of: server

<keepalive-select-enable> enables the select manager for keepalives. The select manager is a Resin Professional feature allowing more keepalives by detaching threads from sockets.

Normally, this should be left enabled.**default:** true

24.7.12 <keepalive-select-thread-timeout>

child of: server

<keepalive-select-thread-timeout> is a short timeout allowing the select manager to wait for a keepalive before detaching the thread. This value would not normally be changed.d wou**default:** 1s

24.7.13 <keepalive-timeout>

child of: server

<keepalive-timeout> configures how long a keepalive connection should wait for a new request before closing.

Keepalives are used both for HTTP connections and for load-balancing and clustering connections. HTTP connections generally have a single HTML page, followed by a number of image requests. By using keepalives, all the requests can use a single socket. The <keepalive-timeout> should be long enough to catch all the HTTP burst requests, but can close after the burst is complete. A value of 5s or 10s is generally sufficient.

The load-balancing and clustering keepalives have a different timeout behavior. Since load-balancing sockets are reused for multiple clients, they can have longer timeouts.**default:** 10s

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server-default>
      <http port="80"/>

      <thread-max>512</thread-max>

      <keepalive-max>100</keepalive-max>
      <keepalive-timeout>10s</keepalive-timeout>
    </server-default>

    <server id="web-a" address="192.168.0.10"/>
       ...
  </cluster>
</resin>
```

keepalive-thread-max in resin.xml

24.7.14 <load-balance-connect-timeout>

child of: server

<load-balance-connect-timeout> configures the maximum time a client connection to a cluster-port should take. The load balance and persistent sessions use load-balance-connect-timeout to connect to backend or peer servers in the cluster.

Lower values detect failed servers more quickly, but a too-low value can timeout too quickly for a live server with some network congestion.**default:** 5s

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="app-tier">
        <server-default>
          <load-balance-connect-timeout>2s</load-balance-connect-timeout>
        </server-default>

        <server id="a" address="192.168.0.10" port="6800"/>
        <server id="b" address="192.168.0.11" port="6800"/>

        <host id="">
          ...
    </cluster>
</resin>
```

load-balance-connect-timeout

24.7.15 <load-balance-recover-time>

child of: server

<load-balance-recover-time> is the maximum time the load balancer will consider the server dead after a failure before retrying the connection.

Resin uses the load-balance-recover-time to avoid wasting time trying to connect to an unavailable app-tier server.

Lower values let the load balancer use a restarted app-tier server faster, but lower values also increase the overhead of trying to contact unavailable servers.**default:** 15s

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="app-tier">
        <server-default>
          <load-balance-recover-time>10s</load-balance-recover-time>
        </server-default>

        <server id="a" address="192.168.0.10" port="6800"/>
        <server id="b" address="192.168.0.11" port="6800"/>

        <host id="">
          ...
    </cluster>
</resin>
```

load-balance-recover-time

24.7.16 <load-balance-idle-time>

child of: server

<load-balance-idle-time> is the maximum time the load balancer and distributed sessions will leave an idle socket before closing it.

The default value is normally sufficient, since it tracks the keepalive of the cluster port.

load-balance-idle-time must be less than the `keepalive` value of the target cluster-port.

The load balancer and distributed sessions reuse sockets to the cluster peer and app-tier servers to improve TCP performance. The load-balance-idle-time limits the amount of time those sockets can remain idle.

Higher values may improve the socket pooling, but may also increase the chance of connecting to a closed server.**default:** keepalive-timeout - 1s

24.7.17 <load-balance-warmup-time>

child of: server

The time the load balancer uses to throttle connections to an app-tier server that's just starting up.

Java web-applications often start slowly while they initialize caches. So a newly-started application will often be slower and consume more resources than a long-running application. The warmup-time increases Resin's reliability by limiting the number of requests to a new app-tier server until the server has warmed up.

Larger values give the application a longer time to warm up.**default:** 60s

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="app-tier">
        <server-default>
          <load-balance-warmup-time>60s</load-balance-warmup-time>
        </server-default>

        <server id="a" address="192.168.0.10" port="6800"/>
        <server id="b" address="192.168.0.11" port="6800"/>

        <host id="">
          ...
    </cluster>
</resin>
```

load-balance-warmup-time

24.7.18 <load-balance-weight>

child of: server

load-balance-weight assigns a load-balance weight to a backend server. Servers with higher values get more requests. Servers with lower values get fewer requests.

In some cases, some app-tier servers may be more powerful than others. load-balance-weight lets the load-balancer assign more connections to the more powerful machines.

Test and profiling servers can also use load-balance-weight to receive a small number of connections for profiling purposes.

Larger values tell the load-balancer to assign more requests to the app-tier server.**default:** 100

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="app-tier">
        <server id="a" address="192.168.0.10" port="6800"/>
        <server id="b" address="192.168.0.10" port="6800"/>

        <server id="test" address="192.168.0.100" port="6800">
            <load-balance-weight>1</load-balance-weight>
        </server>

        <host id="">
          ...
    </cluster>
</resin>
```

load-balance-weight

24.7.19 <memory-free-min>

child of: server

<memory-free-min> improves server reliability by detecting low-memory situations caused by memory leaks and forcing a clean server restart. Since Resin's watchdog service reliably restarts the server, a website can improve stability by forcing a restart before memory becomes a major problem. The memory-free-min restart will also log a warning, notifying the developers that a potential memory leak needs to be resolved.

When free heap memory gets very low, the garbage collector can run continually trying to free up extra memory. This continual garbage collection can send the CPU time to 100%, cause the site to become completely unresponsive, and yet take a long time before finally failing to an out of memory error (forcing an unclean restart). To avoid this situation, Resin will detect the low-memory condition and gracefully restart the server when free memory becomes too low.

The ultimate solution to any memory leak issues is to get a memory profiler, find the leaking memory and fix it. <memory-free-min> is just a temporary bandage to keep the site running reliably until the memory leak can be found and fixed.**default:** 1m

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server-default>
      <thread-max>512</thread-max>

      <memory-free-min>1m</memory-free-min>
    </server-default>

    <server id="web-a" address="192.168.0.10"/>
        ...
  </cluster>
</resin>
```

memory-free-min resin.xml

24.7.20 <port>

child of: server

The server <port> defines the TCP port for Resin cluster communication and load balancing. Most server instances will use a common port like 6800, while machines with multiple servers may use multiple ports like 6800 and 6801.**default:** 6800

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">
    <server id="app-a" address="192.168.1.11" port="6800"/>
    <server id="app-b" address="192.168.1.11" port="6801"/>

    <server id="app-b" address="192.168.2.12" port="6800"/>
    <server id="app-c" address="192.168.2.12" port="6801"/>

    ...
  </cluster>
</resin>
```

multipl servers on a machine

24.7.21 <protocol>

child of: server

<protocol> configures custom socket protocols using Resin's thread and connection management.

The custom protocol will extend from com.caucho.server.port.Protocol.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="web-tier">

  <server id="a">
    <protocol address="localhost" port="8888">
      <type>example.Magic8BallProtocol</type>
    </port>
  </server>

</cluster>
</resin>
```

24.7.22 <server>

child of: cluster

<server> configures a JVM instance in the cluster. Each <server> is uniquely identified by its `id` attribute. The `id` will match the -server-id command line argument.

The server listens to an internal network address, e.g. 192.168.0.10:6800 for clustering, load balancing, and administration.

The current server is managed with a ServerMXBean. The ObjectName is `resin:type=Server` .

Peer servers are managed with ServerConnectorMXBean. The ObjectName is `resin:type=ServerConnector,name=server-id` .

id	unique server identifier	required
address	IP address of the cluster port	127.0.0.1
port	The cluster port	6800

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="web-tier">
        <server id="a" address="192.168.0.10" port="6800">
          <http port="8080"/>
        </server>

        <server id="b" address="192.168.0.11" server-port="6800">
          <http port="8080"/>
        </server>

        <server id="c" address="192.168.0.12" server-port="6800">
          <http port="8080"/>
        </server>

        <host id="">
          ...
    </cluster>
</resin>
```

server

Main configuration for the server, configuring ports, threads and virtual hosts.

- Common resources for all virtual hosts and web-apps.
- Thread pooling
- HTTP and SRUN/Cluster ports
- Caching
- virtual host configuration and common web-app-default

The <server> will generally contain a <class-loader> configuration which loads the resin/lib jars dynamically, allowing for system-wide jars to be dropped into resin/lib. <server> configures the main dynamic environment. Database pools common to all virtual hosts, for example, should be configured in the <server> block.

The <server> configures the <thread-pool> and a set of <http> and <srun> ports which share the thread pool. Requests received on those ports will use worker threads from the thread pool.

alternate-session-url-prefix	a prefix to add the session to the beginning of the URL as a path prefix instead of the standard ;jsessionid= suffix. For clients like mobile devices with limited memory, this will allow careful web designers to minimize the page size.	null
keepalive-max	the maximum number of keepalive connections	512
keepalive-thread-timeout	the maximum time a connection is maintained in the keepalive state	120s

```
<server>
  ...
  <alternate-session-url-prefix>/~J=</alternate-session-url-prefix>
  ...
```

alternate-session-url-prefix

EL variables and functions

VARIABLE	CORRESPONDING API
serverId	*server*.getServerId()
root-dir	*server*.getRootDirectory()
server-root	*server*.getRootDirectory()

EL variables defined by <server>

FUNCTION	CORRESPONDING API
jndi	Jndi.lookup(String)

EL functions defined by <server>

24.7.23 <server-default>

child of: cluster

<server-default> defines default values for all <server> instances. Since most <server> configuration is identical for all server instances, the shared configuration belongs in a <server-default>. For example, <http> ports, timeouts, JVM arguments, and keepalives are typically identical for all server instances and therefore belong in a server-default.

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="web-tier">
        <server-default>
           <thread-max>512</thread-max>

           <jvm-arg>-Xmx512m -Xss1m</jvm-arg>

           <http port="8080"/>
        </server-default>

        <server id="a" address="192.168.0.10" port="6800"/>
        <server id="b" address="192.168.0.11" port="6800"/>
        <server id="c" address="192.168.0.12" port="6800"/>

        <host id="">
          ...
    </cluster>
</resin>
```

server

24.7.24 <shutdown-wait-max>

child of: server

<shutdown-wait-max> configures the maximum time the server will wait for the graceful shutdown before forcing an exit.**default:** 60s

24.7.25 <socket-timeout>

child of: http, cluster-port, protocol, server

<socket-timeout> is the maximum time a socket load balancer and distributed sessions will wait for a read or write to a cluster socket.

Crashed servers may never respond to a read request or accept a write. The socket-timeout lets Resin recover from these kinds of crashes.

Lower values can detect crashes more quickly, but too-low values may report bogus failures when the server machine is just a little slow.**default:** 60s

```
<resin xmlns="http://caucho.com/ns/resin">
    <cluster id="app-tier">
        <server-default>
          <socket-timeout>60s</socket-timeout>
        </server-default>

        <server id="a" address="192.168.0.10" port="6800"/>
        <server id="b" address="192.168.0.11" port="6800"/>

        <host id="">
          ...
    </cluster>
</resin>
```

socket-timeout

24.7.26 <thread-idle-max>

child of: server

<thread-idle-max> configures the maximum number of idle threads in the thread pool. <thread-idle-max> works with <thread-idle-min> to maintain a steady number of idle threads, avoiding the creation or destruction threads when possible.

<thread-idle-max> should be set high enough beyond <thread-idle-min> so a spiky load will avoid creating a thread and then immediately destroying it.**default:** 10

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server-default>
      <thread-max>512</thread-max>

      <thread-idle-min>10</thread-idle-min>
      <thread-idle-max>20</thread-idle-max>

      <jvm-arg>-Xss1m -Xmx1024m</jvm-arg>
    </server-default>

    <server id="web-a" address="192.168.0.10"/>
        ...
  </cluster>
</resin>
```

thread-idle-max in resin.xml

24.7.27 <thread-idle-min>

child of: server

<thread-idle-min> configures the minimum number of idle threads in the thread pool. <thread-idle-min> helps spiky loads, avoiding delays for thread requests by keeping threads ready for future requests. When the number of idle threads drops below <thread-idle-min>, Resin creates a new thread.

<thread-idle-min> should be set high enough to deal with load spikes. Since idle threads are relatively inexpensive in modern operating systems, having a number of idle threads is not a major resource hog, especially since these threads are idle, waiting for a new job.

<thread-idle-min> works together with <thread-idle-max> to avoid thread allocation thrashing, i.e. avoiding creating a new thread because of <thread-idle-min> and then quickly destroying it because of <thread-idle-max>.**default:** 5

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server-default>
      <thread-max>512</thread-max>

      <thread-idle-min>10</thread-idle-min>
      <thread-idle-max>20</thread-idle-max>

      <jvm-arg>-Xss1m -Xmx1024m</jvm-arg>
    </server-default>

    <server id="web-a" address="192.168.0.10"/>
       ...
  </cluster>
</resin>
```

thread-idle-min in resin.xml

24.7.28 <thread-max>

child of: server

<thread-max> configures the maximum number of threads managed by Resin's thread pool. Resin's thread pool is used for connection threads, timers, and Resin worker threads for JMS, JCA and EJB. Since Resin's thread pool only manages Resin threads, the actual number of threads in the JVM will be higher.

Modern operating systems can handle a fairly large number of threads, so values of 512 or 1024 are often reasonable values for thread-max. The main limitation for thread-max is virtual memory. Since each thread takes up stack space (configured with -Xss), a 32-bit system might have a thread limit based on virtual memory.

For example, on Linux the user space is only 2G. If the heap memory is 1024m (-Xmx1024m) and the stack size is 1m (-Xss1m), the maximum number of threads is somewhat less than 1024.

In general, JVMs do not handle running out of threads very well, either freezing or throwing out of memory errors. Although it may be necessary to limit the number of threads to avoid running out of memory, <thread-max> should generally be set to a high value.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="web-tier">
    <server-default>
      <thread-max>512</thread-max>

      <jvm-arg>-Xss1m -Xmx1024m</jvm-arg>
    </server-default>

    <server id="web-a" address="192.168.0.10"/>
       ...
  </cluster>
</resin>
```

thread-max in resin.xml

default: 4096

24.7.29 <user-name>

child of: server

<user-name> configures the operating system user Resin should run as. Since the HTTP port 80 is protected in Unix, the web server needs to start as root to bind to port 80. For security, Resin should switch to a non-root user after binding to port 80.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">

    <server-default>
      <http port="80"/>

      <user-name>resin</user-name>
    </server-default>

    <server id="web-a"/>
    ...
  </cluster>
</resin>
```

resin.xml with user-name

24.7.30 <watchdog-arg>

child of: server

The <watchdog-arg> configures arguments for the watchdog process. The watchdog improves reliability by monitoring the Resin instance, restarting it if necessary.

The <watchdog-arg> typically is used to enable jconsole for the watchdog JVM.

```
<resin xmlns="http://caucho.com/ns/resin">
  <cluster id="app-tier">
    <server-default>

      <watchdog-arg>-Dcom.sun.management.jmxremote</watchdog-arg>

    </server-default>

    ...

  </cluster>
</resin>
```

resin.xml

24.7.31 <watchdog-port>

child of: server

<watchdog-port> configures the administration port for the watchdog JVM. The watchdog launches the server JVM and monitors its health, restarting the JVM when necessary to improve site reliability. The command line arguments use the watchdog-port for the "start" and "stop" commands to tell the watchdog to start and stop the server JVM. The administration also uses the watchdog port for watchdog administration.

The watchdog port will use the same <address> as the server, so it will always be an internal network address, never an external internet address.**default:** 6700

24.8 Session tags

24.8.1 <cookie-domain>

child of: session-config

<cookie-domain> configures the host domain to use for sessions, i.e. value of the Domain cookie attributes.

By default, browsers only send cookies back to the exact host which sent them. In some virtual host cases, it makes sense to share the same cookie value for multiple virtual hosts in the same domain. For example, `caucho.com` might want a single cookie to be used for both `gryffindor.caucho.com` and `slytherin.caucho.com` . In this case the `cookie-domain` would be set to `caucho.com` .

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="app-tier">

  <host id="caucho.com">
    <host-alias>gryffindor.caucho.com</host-alias>
    <host-alias>slytherin.caucho.com</host-alias>

    <web-app-default>

      <session-config cookie-domain="caucho.com"/>

    </web-app-default>
  </host>

</cluster>
</resin>
```

cookie-domain for caucho.com

24.8.2 <cookie-length>

child of: session-config

<cookie-length> sets the length of the generated cookie value. In some rare cases, the cookie-length might need to be shortened or the cookie-length might be extended to add randomness to the cookie value. For the most part, this value should be left alone.**default:** 14

24.8.3 <cookie-max-age>

child of: session-config

<cookie-max-age> sets how long the browser should keep the session cookie.

By default, browsers keep cookies as long as they are open. When the browser is restarted, the cookies are dropped. cookie-max-age tells the browser to keep the cookie for a certain time period. Applications can make this time "infinite" by setting a large number.

24.8.4 <cookie-version>

<cookie-version> sets the version of the cookie spec for sessions.**default:** 1.0

24.8.5 <ignore-serialization-errors>

child of: session-config

<ignore-serialization-errors> is used with persistent sessions in certain rare cases where some session data is serializable and other data is not serializable. <ignore-serialization-errors> simply skips session values which do not implement `java.io.Serializable` when serializing the session.**default:** false

24.8.6 <session-config>

Session configuration parameters.

session-timeout	The session timeout in minutes, 0 means never timeout.	30 minutes

Servlet 2.4 definition for session-timeout

Resin adds a number of `session-config` tags.

always-load-session	Reload data from the store on every request. (resin 1.2)	false
always-save-session	Save session data to the store on every request. (resin 1.2)	false
cookie-version	Version of the cookie spec for sessions. (resin 1.2)	1.0
cookie-domain	Domain for session cookies. (resin 1.2)	none
cookie-max-age	Max age for persistent session cookies. (resin 2.0)	none
cookie-length	Maximum length of the cookie. (resin 2.1.1)	Integer.MAX_VALUE
enable-cookies	Enable cookies for sessions. (resin 1.1)	true
enable-url-rewriting	Enable URL rewriting for sessions. (resin 1.1)	true
ignore-serialization-errors	When persisting a session, ignore any values which don't implement java.io.Serializable	false
invalidate-after-listener	Invalidate the session after notifying session listeners.	false
reuse-session-id	Reuse the session id even if the session has timed out. (resin 2.0.4)	true
save-only-on-shutdown	Only save session when the application shuts down. (resin 1.2.3)	false

ssl-cookie-name	Set a different cookie name to use for SSL connections, and add the "secure" flag when setting the cookie in the browser.	
session-max	Maximum active sessions	4096
use-persistent-store	Uses the current persistent-store to save sessions. (resin 3.0.8)	none

Resin extensions to session-config

By default, both `enable-cookies` and `enable-url-rewriting` are true. To force url rewriting, you would create a configuration like:

```
<web-app id='/'>

  <session-config
   enable-cookies='false'
   enable-url-rewriting='true'/>

</web-app>
```

The `session-timeout` and `session-max` are usually used together to control the number of sessions. Sessions are stored in an LRU cache. When the number of sessions in the cache fills up past `session-max` , the oldest sessions are recovered. In addition, sessions idle for longer than `session-timeout` are purged.

```
<web-app id='/dir'>

  <session-config>
     <!-- 2 hour timeout -->
     <session-timeout>120</session-timeout>
     <session-max>4096</session-max>
  </session-config>

</web-app>
```

using session-config and session-timeout to control the number of sessions

`cookie-length` is used to limit the maximum length for the session's generated cookie for special situations like WAP devices. Reducing this value reduces the randomness in the cookie and increases the chance of session collisions.

`reuse-session-id` defaults to true so that Resin can share the session id amongst different web-apps.

The class that corresponds to <session-config> is

24.8.7 <session-max>

child of: session-config

<session-max> sets the maximum number of sessions stored in memory for a particular <web-app>. The total number of persisted sessions may be larger.**default:** 4096

24.8.8 <save-mode>

child of: session-config

<save-mode> configures when Resin should save a persistence session during a request. The values are:

after-request	Save the session after the request has been served and completed
before-headers	Save the session before sending headers to the browser
on-shutdown	Only save the session when Resin is shutting down

In some situations, like redirects, a fast browser can send a request back to Resin before the session is persisted with the `after-request` save-mode. If the server is configured without sticky sessions, the load balancer might send the request to a different server, which may not get the updated session. In the situation, either the save-mode should be changed to `before-headers` or sticky sessions should be enabled.

If the save-mode is `before-headers` , the application should take care to make any session changes before sending data to the browser.**default:** after-request

24.8.9 <session-timeout>

child of: session-config

<session-timeout> sets how long a <web-app> should keep an idle session before invalidating it. The value is specified in minutes.**default:** 30min

24.9 EL Variables and Functions

Examples

Servlet/Filter initialization

EL expressions can be used to configure servlets and filters. `init-param` values can use JSP EL expressions, including the ability to use system properties.

Servlets, filters, and resources can be configured like beans with setter methods are called directly (See Bean-style init).

One example use for the bean-style servlet initialization is to avoid JNDI lookup inside the servlet code. For example, a servlet that that uses a JDBC DataSource might look like:

```
package test;

...

public class TestServlet extends HttpServlet {
  private DataSource _dataSource;

  /**
   * Bean setter is called to configure the servlet
   * before the init() method.
   */
  public void setDataSource(DataSource dataSource)
  {
    _dataSource = dataSource;
  }

  ...
}
```

Servlet using JDBC

The servlet is configured as follows:

```
<web-app>
  <allow-servlet-el/>

  <servlet servlet-name='test'
           servlet-class='test.TestServlet'>
    <init>
      <data-source>${jndi("java:comp/env/jdbc/test")}</data-source>
    </init>
  </servlet>

  ...
</web-app>
```

Example configuration

The <`data-source`> xml tag corresponds to the `setDataSource` method of the bean. More infomation on this powerful pattern is in the bean configuration section of the documentation.

24.9.1 Environment variables

Environment variables inherited by the process from the operting system are available as variables in el expressions, for example `${LANG}` .

24.9.2 fmt.sprintf()

Format a string using a sprintf-like format string.

```
fmt.sprintf(format[,arg1, arg2 ... argN])
```

format	the format string (see below)	required
arg1..argN	the values used for the conversions in the format string	n/a

`sprintf` accepts a series of arguments, applies to each a format specifier from 'format', and returns the formatted data as a string. 'format' is a string containing two types of objects: ordinary characters (other than '%'), which are copied unchanged to the output, and conversion specifications, each of which is introduced by '%'. (To include '%' in the output, use '%%' in the format string).

A conversion specification has the following form:

```
%[FLAGS][WIDTH][.PREC][TYPE]
```

TYPE is required, the rest are optional.

The following TYPE's are supported:

%%	a percent sign
%c	a character with the given number
%s	a string, a null string becomes "#null"
%z	a string, a null string becomes the empty string ""
%d	a signed integer, in decimal
%o	an integer, in octal
%u	an integer, in decimal
%x	an integer, in hexadecimal
%X	an integer, in hexadecimal using upper-case letters
%e	a floating-point number, in scientific notation
%E	a floating-point number, like %e with an upper-case "E"
%f	a floating-point number, in fixed decimal notation
%g	a floating-point number, in %e or %f notation

%G	a floating-point number, like %g with an upper-case "E"
%p	a pointer (outputs a value like the default of toString())

Intepret the word 'integer' to mean the java type long. Since java does not support unsigned integers, all integers are treated the same.

The following optional FLAGS are supported:

0	If the TYPE character is an integer leading zeroes are used to pad the field width instead of spaces (following any indication of sign or base).
+	Include a '+' with positive numbers.
(a space)	use a space placeholder for the '+' that would result from a positive number
-	The result of is left justified, and the right is padded with blanks until the result is 'WIDTH' in length. If you do not use this flag, the result is right justified, and padded on the left.
#	an alternate display is used, for 'x' and 'X' a non-zero result will have an "0x" prefix; for floating point numbers the result will always contain a decimal point.
j	escape a string suitable for a Java string, or a CSV file. The following escapes are applied: " becomes \", newline becomes \n, return becomes \r, \ becomes \\.
v	escape a string suitable for CSV files, the same as 'j' with an additional " placed at the beginning and ending of the string
m	escape a string suitable for a XML file. The following escapes are applied: < becomes <, > becomes > & becomes & ' becomes &#039, " becomes &034;

The optional WIDTH argument specifies a minium width for the field. Spaces are used unless the '0' FLAG was used to indicate 0 padding.

The optional PREC argument is introduced with a '.', and gives the maximum number of characters to print; or the minimum number of digits to print for integer and hex values; or the maximum number of significant digits for 'g' and 'G'; or the number of digits to print after the decimal point for floating points.

24.9.3 fmt.timestamp()

Format a timestamp string.

```
fmt.timestamp(format[,date])
```

format	the format string (see below)	required
date	an object with or or	the current date and time

```
msg="The current date and time is ${fmt.timestamp('%Y/%m/%d %H:%M:%S.%s')}"
msg="time=${fmt.timestamp('[%Y/%m/%d %H:%M:%S.%s]')}"
```

`format` contains regular characters, which are just copied to the output string, and percent codes which are substituted with time and date values.

CODE	**MEANING**
%a	day of week (short)
%A	day of week (verbose)
%b	day of month (short)
%B	day of month (verbose)
%c	Java locale date
%d	day of month (two-digit)
%H	24-hour (two-digit)
%I	12-hour (two-digit)
%j	day of year (three-digit)
%m	month (two-digit)
%M	minutes
%p	am/pm
%S	seconds
%s	milliseconds
%W	week in year (three-digit)

%w	day of week (one-digit)
%y	year (two-digit)
%Y	year (four-digit)
%Z	time zone (name)
%z	time zone (+/-0800)

24.9.4 host

VARIABLE	MEANING
name	The name of the host
regexp	Regular expression values for host regexp matches
root	The root directory of the host
url	The canonical url of the host

host properties

Example

```
<host regexp="www.([^.]+).com">
  <root-directory>/opt/www/${host.regexp[1]}</root-directory>

  <context-param server-id="${server.name}"/>

  <web-app id="/">
    <document-directory>webapps/ROOT</document-directory>
  </web-app>
</host>
```

24.9.5 java

VARIABLE	MEANING
version	The JDK version

java properties

24.9.6 jndi()

The configuration EL supports the static function jndi:lookup. jndi:lookup can be used to lookup a JNDI value for the configuration.

```
<servlet servlet-name='foo'
         servlet-class='qa.FooServlet'>
  <init>
    <data-source>${jndi:lookup("java:comp/env/jdbc/test")}</data-source>
  </init>
</servlet>
```

configuring JNDI

24.9.7 resin

VARIABLE	MEANING
address	The local IP address
conf	Path to the configuration file
home	The the location of the Resin executables
hostname	The local hostname as returned by InetAddress
root	The location of the content, specified at startup with –resin-root
serverId	The identity of the active <server>, specified at startup with –server
version	The resin version, e.g. 3.1.0

resin properties

24.9.8 server

Values related to the active <server>.

VARIABLE	MEANING
address	the bind address of the cluster and load balancing port
id	The identity of the active <server>, specified at startup with –server
port	the cluster and load balancing port
httpAddress	the bind address of the http listener, INADDR_ANY for all addresses
httpPort	the port number of the http listener
httpsAddress	the bind address of the ssl http listener, INADDR_ANY for all addresses
httpsPort	the port number of the ssl http listener

server properties

24.9.9 System properties

System properties are available as variables in el expressions. Many system property names are not valid el identifiers; in that case the `system` variable is used, for example `${system['java.io.tmpdir']}` .

A full list of standard java system properties is provided in the javadoc for java.lang.System.

VARIABLE	MEANING
java.io.tmpdir	Default temp file path
os.name	Operating system name
os.arch	Operating system architecture
os.version	Operating system version
user.name	User's account name
user.home	User's home directory
user.dir	User's current working directory

Standard system properties

24.9.10 webApp

VARIABLE	MEANING
name	The name of the web-app
contextPath	The context path of the web-app
regexp	Regular expression values for web-app regexp matches
root	The root directory of the web-app
url	The canonical url of the web app

webApp properties

24.10 Web Application: tags

See Also

- Resources tags for beans, components, databases, @Stateless and @Stateful EJBs, JMS queues, etc.
- Security for a full description of Resin's authentication and authorization.
- Rewrite tags for Resin's rewrite-dispatch rewriting.

24.10.1 <access-log>

child of: web-app

<access-log> configures a HTTP access log for an indivitual web-app. See access-log in the <host> tag for more information.

24.10.2 <active-wait-time>

child of: web-app

<active-wait-time> sets a 503 busy timeout for requests trying to access a restarting web-app. If the timeout expires before the web-app complete initialization, the request will return a 503 Busy HTTP response.

```
element active-wait-time {
  r_period-Type
}
```

24.10.3 <allow-servlet-el>

child of: web-app

The <allow-servlet-el> flag configures whether <servlet> tags allow EL expressions in the init-param.

```
element allow-servlet-el {
  r_boolean-Type
}
```

24.10.4 <archive-path>

child of: web-app

<archive-path> configures the location of the web-app's .war file. In some configurations, the .war expansion might not use the `webapps/` directory, but will still want automatic war expantion.

```
element archive-path {
  r_path-Type
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

  <host id="">

    <web-app id="/foo"
             root-directory="/var/www/foo"
             archive-path="/var/www/wars/foo.war"/>

  </host>
</cluster>
</resin>
```

Example: resin.xml explicit archive location

24.10.5 <auth-constraint>

child of: security-constraint

Requires that authenticated users fill the specified role. In Resin's JdbcAuthenticator, normal users are in the "user" role. Think of a role as a group of users.

The roles are defined by the authenticators (see Resin security). When using Resin's <management> authentication as a default, the role name is `resin-admin` . (See Resin management.)

ATTRIBUTE	DESCRIPTION
role-name	Roles which are allowed to access the resource.

```
element auth-constraint {
  description*,

  role-name*
}
```

The following example protects the /admin subdirectory of a web-app by requiring a user to logon with Resin's <management> users, i.e. using the same requirement as /resin-admin.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <login uri="basic:"/>

  <security-constraint url-pattern="/admin/*">
    <auth-constraint role-name="resin-admin"/>
  </security-constraint>

</web-app>
```

Example: WEB-INF/resin-web.xml auth constraints

24.10.6 <cache-mapping>

child of: web-app

<cache-mapping> specifies `max-age` and `Expires` times for cacheable pages. See caching for more information.

<cache-mapping> is intended to provide Expires times for pages that have ETags or Last-Modified specified, but do not wish to hard-code the max-age timeout in the servlet. For example, Resin's FileServlet relies on cache-mapping to set the expires times for static pages. Using cache-mapping lets cacheable pages be configured in a standard manner.

<cache-mapping> does not automatically make pages cacheable. Your servlets must already set the `ETag` (or Last-Modified) header to activate <cache-mapping>.

ATTRIBUTE	DESCRIPTION	DEFAULT
expires	A time interval to be used for the HTTP Expires header.	
max-age	A time interval to be used for the "Cache-Control max-age=xxx" header. max-age affects proxies and browsers.	
s-max-age	A time interval to be used for the "Cache-Control s-max-age=xxx" header. s-max-age affects proxy caches (including Resin), but not browsers.	
url-pattern	A pattern matching the url: `/foo/*` , `/foo` , or `*.foo`	

url-regexp	A regular expression matching the url

The time interval defaults to seconds, but will allow other periods:

SUFFIX	MEANING
s	seconds
m	minutes
h	hours
D	days

```
element cache-mapping {
  (url-pattern | url-regexp)
  & expires?
  & max-age?
  & s-max-age?
}
```

- cache-mapping requires an enabled <cache>. If the cache is disabled, cache-mapping will be ignored.
- cache-mapping does not automatically make a page cacheable. Only cacheable pages are affected by cache-mapping, i.e. pages with an ETag or Last-Modified.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <cache-mapping url-pattern='/*'
                 max-age='10'/>

  <cache-mapping url-pattern='*.gif'
                 max-age='15m'/>

</web-app>
```

Example: cache-mapping in resin-web.xml

24.10.7 <constraint>

child of: security-constraint

Defines a custom constraint. Applications can define their own security constraints to handle custom authentication requirements.

ATTRIBUTE	DESCRIPTION
resin:type	A class that extends AbstractConstraint
init	initialization parameters, set in the object using Bean-style setters and getters

```
element constraint {
  class
  & init?
}
```

24.10.8 <context-param>

child of: web-app

Initializes application (ServletContext) variables. `context-param` defines initial values for `application.getInitParameter("foo")` . See also ServletContext.getInitParameter().

ATTRIBUTE	DESCRIPTION
param-name	Named parameter
param-value	Parameter value
`foo`	Parameter name

<context-param attributes

```
element context-param {
  (param-name, param-value)*
  | (attribute * { string })*
  | (element * { string })*
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

  <context-param>
    <param-name>baz</param-name>
    <param-value>value</param-value>
  </context-param>

  <!-- shortcut -->
  <context-param foo="bar"/>

</web-app>
```

Example: context-param in resin-web.xml

24.10.9 <cookie-http-only>

child of: web-app

The <cookie-http-only> flag configures the Http-Only attribute for all Cookies generated from the web-app. The Http-Only attribute can add security to a website by not forwarding HTTP cookies to SSL HTTPS requests.

```
element cookie-http-only {
  r_boolean-Type
}
```

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">

  <host id="www.foo.com">
    <web-app id="" root-directory="/var/www/foo">
      <cookie-http-only>true</cookie-http-only>
    <web-app id="">
  </host>

  <host id="www.foo.com:443">
    <web-app id="" root-directory="/var/www/foo-secure">
      <secure/>
    <web-app id="">
  </host>

</cluster>
</resin>
```

Example: <cookie-http-only> in resin.xml

24.10.10 <ear-deploy>

child of: host, web-app

Specifies ear expansion.

ear-deploy can be used in web-apps to define a subdirectory for ear expansion.

ATTRIBUTE	DESCRIPTION	DEFAULT
archive-path	The path to the directory containing ear files	path
ear-default	resin.xml default configuration for all ear files, e.g. configuring database, JMS or EJB defaults.	
expand-cleanup-fileset	Specifies the files which should be automatically deleted when a new .ear version is deployed.	
expand-directory	directory where ears should be expanded	value of `path`
expand-prefix	automatic prefix of the expanded directory	_ear_
expand-suffix	automatic suffix of the expanded directory	
lazy-init	if true, the ear file is only started on first access	false
path	The path to the deploy directory	required
redeploy-mode	"automatic" or "manual". If automatic, detects new .ear files automatically and deploys them.	automatic
url-prefix	optional URL prefix to group deployed .ear files	

<ear-deploy> Attributes

```
element ear-deploy {
  path
  & archive-directory?
  & ear-default?
  & expand-cleanup-fileset?
  & expand-directory?
  & expand-path?
  & expand-prefix?
  & expand-suffix?
  & lazy-init?
  & redeploy-mode?
  & require-file*
  & url-prefix?
}
```

24.10.11 <error-page>

child of: web-app

ATTRIBUTE	DESCRIPTION
error-code	Select the error page based on an HTTP status code
exception-type	Select the error page based on a Java exception
location	The error page to display

ATTRIBUTE	TYPE
javax.servlet.error.status_code	java.lang.Integer
javax.servlet.error.message	java.lang.String
javax.servlet.error.request_uri	java.lang.String
javax.servlet.error.servlet_name	java.lang.String
javax.servlet.error.exception	java.lang.Throwable
javax.servlet.error.exception_type	java.lang.Class

Request attributes for error handling

```
element error-page {
  (error-code | exception-type)?
  & location
}
```

By default, Resin returns a 500 Servlet Error and a stack trace for exceptions and a simple 404 File Not Found for error pages. Applications can customize the response generated for errors.

```
<web-app xmlns="http://caucho.com/ns/resin">
  <error-page>
    <error-code>404</error-code>
    <location>/file_not_found.jsp</location>
  </error-page>
</web-app>
```

Example: Catching File Not Found

```
<web-app xmlns="http://caucho.com/ns/resin">
   <error-page exception-type="java.lang.NullPointerException"
               location="/nullpointer.jsp"/>
</web-app>
```

Example: Catching Exceptions

The error page can use request attributes to obtain information about the request that caused the error:

```
<%@ page session="false" isErrorPage="true" %>

<html>
<head><title>404 Not Found</title></head>
<body>
<h1>404 Not Found</h1>

The url <code>${requestScope["javax.servlet.error.request_uri"]}</code>
was not found.
</body>
</html>
```

Example: /file_not_found.jsp

24.10.12 <filter>

Defines a filter name for later mapping. Because Filters are fully integrated with Resin-IoC, they can use dependency-injection, transactional aspects, custom interception with @InterceptorType, and event handling with @Observes.

ATTRIBUTE	DESCRIPTION
filter-name	The filter's name (alias)
filter-class	The filter's class (defaults to filter-name), which extends javax.servlet.Filter
init	Resin-IoC initialization configuration, see Resin-IoC
init-param	Initialization parameters, see FilterConfig.getInitParameter

```
element filter {
  filter-name
  & filter-class
  & init*
  & init-param*
}
```

The following example defines a filter alias 'image'

```
<web-app xmlns="http://caucho.com/ns/resin">

  <filter>
    <filter-name>image</filter-name>
    <filter-class>test.MyImage</filter-class>
    <init-param>
      <param-name>title</param-name>
      <param-value>Hello, World</param-value>
    </init-param>
  </filter>

  <filter-mapping>
    <filter-name>image</filter-name>
    <url-pattern>/images/*</url-pattern>
  </filter-mapping>

</web-app>
```

Example: WEB-INF/resin-web.xml

The full Servlet 2.3 syntax for `init-param` is supported as well as a simple shortcut.

```
<web-app id='/'>

<filter filter-name='test.HelloWorld'>
  <init-param foo='bar'/>

  <init-param>
    <param-name>baz</param-name>
    <param-value>value</param-value>
  </init-param>
</servlet>

</web-app>
```

WEB-INF/resin-web.xml

24.10.13 <filter-mapping>

Maps url patterns to filters. `filter-mapping` has two children, `url-pattern` and `filter-name` . `url-pattern` selects the urls which should execute the filter.

`filter-name` can either specify a servlet class directly or it can specify a servlet alias defined by `filter` .

ATTRIBUTE	DESCRIPTION	DEFAULT
dispatcher	REQUEST, INCLUDE, FORWARD, ERROR	REQUEST
filter-name	The filter name	required
url-pattern	A pattern matching the url: `/foo/*` , `/foo` , or `*.foo`	
url-regexp	A regular expression matching the portion of the url that follows the context path	

<filter-mapping> attributes

```
element filter-mapping {
  (url-pattern | url-regexp | servlet-name)+
  & filter-name
  & dispatcher*
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

  <filter>
    <filter-name>test-filter</filter-name>
    <filter-class>test.MyFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>test-filter</filter-name>
    <url-pattern>/hello/*</url-pattern>
  </filter-mapping>

  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>test.HelloWorld</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
</web-app>
```

Example: resin-web.xml filters

`url-regexp` matches the portion of the url that follows the context path. A webapp in `webapps/ROOT` , and a url `http://localhost/foo/hello.html` will have a value of `"/foo/hello.html"` for the purposes of the regular expression match. A webapp in `webapps/baz` and a url `http://localhost/baz/hello.html` will have a url of `"/hello.html"` for the purposes of the regular expression match, because "/baz" is the context path.

24.10.14 <form-login-config>

child of: login-config

Configures authentication using forms. The login form has specific parameters that the servlet engine's login form processing understands. If the login succeeds, the user will see the original page. If it fails, she will see the error page.

ATTRIBUTE	DESCRIPTION	DEFAULT
form-login-page	The page to be used to prompt the user login	
form-error-page	The error page for unsuccessful login	
internal-forward	Use an internal redirect on success instead of a sendRedirect	false
form-uri-priority	If true, the form's j_uri will override a stored URI	false

<form-login-config> attributes

The form itself must have the action `j_security_check` . It must also have the parameters `j_username` and `j_password` . Optionally, it can also have `j_uri` and `j_use_cookie_auth` . `j_uri` gives the next page to display when login succeeds. `j_use_cookie_auth` allows Resin to send a persistent cookie to the user to make following login easier.

`j_use_cookie_auth` gives control to the user whether to generate a persistent cookie. It lets you implement the "remember me" button. By default, the authentication only lasts for a single session.

ATTRIBUTE	DESCRIPTION
j_security_check	The form's mandatory action
j_username	The user name
j_password	The password
j_uri	Optional Resin extension for the successful display page.
j_use_cookie_auth	Optional Resin extension to allow cookie login.

```
element form-login-config {
  form-login-page,
  form-error-page,
  internal-forward,
  form-uri-priority
}
```

The following is an example of a servlet-standard login page:

```
<form action='j_security_check' method='POST'>
<table>
<tr><td>User:<td><input name='j_username'>
<tr><td>Password:<td><input name='j_password'>
<tr><td colspan=2>hint: the password is 'quidditch'
<tr><td><input type=submit>
</table>
</form>
```

Example: login.html

24.10.15 <idle-time>

child of: web-app

The <idle-time> specifies the timeout for lazy-idle web-apps. In some configurations, web-apps are created only on demand and are closed when no requests access the web-app. The idle-time configures when those web-apps should be freed.

For example, the resin-doc web-app uses idle-time for its child web-apps because there are a large number of sub-web-apps for the individual tutorials.

```
element idle-time {
  r_period-Type
}
```

24.10.16 <jsf>

child of: web-app

Configures JSF behavior.

ATTRIBUTE	DESCRIPTION	DEFAULT
fast-jsf	Optimize JSF code generation	true
state-serialization-method	Configures method of JSF state serialization. Setting value to `hessian` provides fast, size optimized Hessian serialization. Method `java` allows fallback to Java Serialization.	hessian
enable-developer-aid	if true, makes snapshots of component tree and request information available via a link displayed at the right bottom corner of a JSF page.	false
developer-aid-link-style	controls appearance of the *JSF Dev Aid* on a JSF page.	position:absolute; bottom:0; right:0

```
element jsf {
  & fast-jsf?
}
```

24.10.17 <jsp>

child of: web-app

Configures JSP behavior.

ATTRIBUTE	DESCRIPTION	DEFAULT
auto-compile	Automatically compile changed JSP files	true
deferred-syntax-allowed-as-literal	enables the #{...} syntax as text contents	true
dependency-check-interval	How often to check the jsp for changes, -1 disables	inherited
el-ignored	Ignore EL expressions in JSP text	false
fast-jstl	Optimize JSTL code generation	true
ignore-el-exception	Ignore exceptions generated in EL expressions. For debugging, set to false	true
is-xml	Default JSP pages to use XML syntax	false
page-encoding	Sets the default page encoding	ISO-8859-1
precompile	Try to load precompiled JSP pages	true
print-null-as-blank	If true, expressions evaluating to null are not printed	false
recompile-on-error	Recompile the JSP file when an Error occurs in loading	false
recycle-tags	Reuse tag instances when possible for performance	true
require-source	Return 404 when JSP source is deleted	false
scriping-invalid	Disables all Java scripting and expression in JSP pages	false
session	Creates sessions for each JSP page	true
static-page-generates-class	If true, JSPs with no active content still generate a .class	true

tld-dir	restricts the directory to scan for .tld files, improving startup performance	WEB-INF
tld-file-set	adds an ant-style pattern for .tld scanning	WEB-INF
trim-directive-whitespace	if true, trims whitespace around JSP directives	false
validate-taglib-schema	if true, validate .tld files against the .tld schema. Set to false to handle invalid .tld files	true
velocity-enabled	if true, velocity-style tags are allowed	false

```
element jsp {
  auto-compile
  & deferred-syntax-allowed-as-literal?
  & dependency-check-interval?
  & el-ignored?
  & fast-jstl?
  & ide-hack?
  & ignore-el-exception?
  & is-xml?
  & page-encoding?
  & precompile?
  & print-null-as-blank?
  & recompile-on-error?
  & recycle-tags?
  & require-source?
  & scripting-invalid?
  & session?
  & static-page-generates-class?
  & tld-dir?
  & tld-file-set?
  & trim-directive-whitespaces?
  & validate-taglib-schema?
  & velocity-enabled?
}
```

24.10.18 <jsp-config>

<jsp-config> configure standard settings for JSP files.

ATTRIBUTE	DESCRIPTION	DEFAULT
url-pattern	selects the URLs which this jsp-config applies to	
el-ignored	If true, EL expressions are ignored	false
page-encoding	Defines the default page encoding for the JSP file	ISO-8859-1
scripting-invalid	If true, Java scripting is forbidden in the JSP page	false
trim-directive-whitespaces	If true, extra whitespace is trimmed around JSP directives	false
is-xml	If true, for XML syntax for JSP pages	false
include-prelude	Includes JSP fragments before the JSP page as headers	
include-coda	Includes JSP fragments before the JSP page as footers	

<jsp-config> Attributes

```
element jsp-config {
  taglib*,
  jsp-property-group*
}

element jsp-property-group {
  url-pattern*,
  deferred-syntax-allowed-as-literal?,
  el-ignored?,
  page-encoding?
  scripting-invalid?
  trim-directive-whitespaces?
  is-xml?
  include-prelude*
  include-coda*
}
```

24.10.19 <lazy-servlet-validate>

default: false

<lazy-servlet-validate> defers validation of servlet classes until the servlet is used. Some servlet libraries are bundled with web.xml files which include servlets with no available classes. Since Resin will normally send an error in this situation, <lazy-servlet-validate> lets you turn the validation off.

```
element lazy-servlet-validate {
  r_boolean-Type
}
```

24.10.20 <listener>

<listener> configures servlet event listeners. The listeners are registered based on interfaces they implement. The listener instances are fully Resin-IoC aware, including dependency injection, observing events, and supporting aspects.

ATTRIBUTE	DESCRIPTION
listener-class	classname of the listener implementation
init	IoC initialization of the listener

<listener> Attributes

INTERFACE	DESCRIPTION
javax.servlet.ServletContextListener	Called when the web-app starts and stops
javax.servlet.ServletContextAttributeI	Called when the web-app attributes change
javax.servlet.ServletRequestListener	Called when the request starts and stops
javax.servlet.ServletRequestAttributeI	Called when request attributes change
javax.servlet.http.HttpSessionListener	Called when HTTP sessions start or stop
javax.servlet.http.HttpSessionAttribut	Called when HTTP sessions attributes change
javax.servlet.http.HttpSessionActivati	Called when HTTP sessions passivate or activate

listener interfaces

```
element listener {
  listener-class,
  init?
}
```

24.10.21 <login-config>

child of: web-app**default:** no authentication

Configures the login method for authentication, one of BASIC, DIGEST or FORM.

See also: Resin security for an overview.

ATTRIBUTE	DESCRIPTION
auth-method	Authentication method, either `BASIC` for HTTP Basic Authentication, `FORM` for form based authentication, or `DIGEST` for HTTP Digest Authentication.
authenticator	Specifies the authenticator to use to lookup users and passwords.
class	Defines a custom class which extends com.caucho.server.security.AbstractLogin
form-login-config	Configuration for form login.
init	Initialization for the custom login class
realm-name	The realm name to use in HTTP authentication

<login-config> Attributes

HTTP Authentication is defined in the RFC HTTP Authentication: Basic and Digest.

HTTP digest authentication is discussed in Digest Passwords.

```
element login-config {
  class?
  & auth-method?
  & authenticator?
  & form-login-config?
  & init?
  & realm-name?
```

24.10.22 <mime-mapping>

child of: web-app

Maps url patterns to mime-types.

ATTRIBUTE	DESCRIPTION
extension	url extension
mime-type	the mime-type

<mime-mapping> attributes

```
element mime-mapping {
  extension,
  mime-type
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

  <mime-mapping>
    <extension>.foo</extension>
    <mime-type>text/html</mime-type>
  </mime-mapping>

  <!-- resin shortcut syntax -->
  <mime-mapping extension='.bar'
                mime-type='text/html'/>

</web-app>
```

Example: WEB-INF/resin-web.xml

Resin has a long list of default mime types in `$RESIN_HOME/conf/app-default.xml`

24.10.23 <multipart-form>

child of: web-app

Enables multipart-mime for forms and file uploads. multipart-mime is disabled by default.

For an uploaded file with a form name of `foo` , the parameter value contains the path name to a temporary file containing the uploaded file. `foo.filename` contains the uploaded filename, and `foo.content-type` contains the content-type of the uploaded file.

ATTRIBUTE	DESCRIPTION	DEFAULT
upload-max	maximum size of an upload request (in kb).	no limit

<multipart-form> Attributes

If the upload is larger than the limit or if multipart-form processing is disabled, Resin will not parse the request and will set an error message in the " `caucho.multipart.form.error` " request attribute. The " `caucho.multipart.form.error.size` " will contain the attempted upload size.

Requests can set the maximum by setting the request attribute " `caucho.multipart.form.upload-max` " with an Integer or Long value.

By default, multipart-form is disabled.

```
element multipart-form {
  enable?
  & upload-max?
}
```

24.10.24 <path-mapping>

child of: web-app

Maps url patterns to real paths. If using a server like IIS, you may need to match the server's path aliases.

ATTRIBUTE	DESCRIPTION
url-pattern	A pattern matching the url: `/foo/*` , `/foo` , or `*.foo`
url-regexp	A regular expression matching the portion of the url that follows the context path
real-path	The prefix of the real path. When used with `url-regexp` , allows substitution variables like `$1` .

<path-mapping> Attributes

```
element path-mapping {
  (url-pattern | url-regexp)

  & real-path
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

<path-mapping url-pattern='/resin/*'
              real-path='e:\resin'/>

<path-mapping url-regexp='/~([^/]*)'
              real-path='e:\home\$1'/>

</web-app>
```

Example: resin-web.xml aliasing paths

24.10.25 <protocol>

child of: servlet, servlet-mapping

<protocol> configures a remoting protocol for a Java bean. The bean is configured with the <servlet> and <servlet-mapping> tags, since it will process HTTP URL requests.

Protocol drivers extend the AbstractProtocolServletFactory interface and can register a URI alias to simplify configuration.

ATTRIBUTE	DESCRIPTION
class	Classname of the protocol driver implementing ProtocolServletFactory
init	Optional IoC initialization for the protocol driver
uri	Protocol configuration shortcut

<protocol> Attributes

URI SCHEME	DESCRIPTION
burlap:	The burlap XML protocol
cxf:	The CXF SOAP implementation
hessian:	The Hessian protocol
xfire:	The XFire SOAP implementation

Current drivers

```
element protocol {
  (class | uri)
  & init?
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

  <servlet-mapping url-pattern="/hello">
                   servlet-class="example.MyHello">
    <protocol uri="hessian:"/>
  </servlet-mapping>

</web-app>
```

Example: Hessian service in resin-web.xml

24.10.26 <redeploy-check-interval>

child of: web-app**default:** 60s

<redeploy-check-interval> specifies how often Resin should check if a .war file has been updated or added to a <web-app-deploy> directory.

```
element redeploy-check-interval {
  r_period-Type
}
```

24.10.27 <redeploy-mode>

child of: web-app**default:** automatic

<redeploy-mode> specifies how Resin handles updates to web-apps and .war files. By default, Resin restarts web-apps when classes or configuration files change.

MODE	DESCRIPTION
automatic	checks for redeployment and auto-redeploy if modified
manual	does not check for redeployment. Only checks if manual (JMX)

```
element redeploy-mode {
  automatic | manual
}
```

24.10.28 Resource Tags

child of: cluster, host, web-app

See the Resource tag documentation for a full list of resources available to the web-app.

All resource tags are available to the <web-app>, like databases, IoC beans and components, EJB stateful, stateless and message beans, JMS queues, remote clients, etc.

```
<web-app xmlns="http://caucho.com/ns/resin">
  <database jndi-name="jdbc/test">
    <driver type="org.postgresql.Driver">
      <url>jdbc:postgresql://localhost/test</url>
      <user>caucho</user>
    </driver>
  </database>
</web-app>
```

Example: web-app database in resin-web.xml

Or IoC-configured Beans:

```
<web-app xmlns="http://caucho.com/ns/resin">
  <bean jndi-name="jdbc/test" type="example.Theater>
    <init name="Balboa Theater">
      <movie title="Attack of the Killer Tomatoes"/>

      <movie title="Snakes on a Plane"/>

      <movie title="Casablanca"/>
    </init>
  </bean>
</web-app>
```

Example: IoC bean in resin-web.xml

24.10.29 <rewrite-dispatch>

child of: cluster, host, web-app

<rewrite-dispatch> defines a set of rewriting rules for dispatching and forwarding URLs. Applications can use these rules to redirect old URLs to their new replacements.

See rewrite-dispatch for more details.

```
<web-app xmlns="http://caucho.com/ns/resin">

 <rewrite-dispatch>
   <dispatch regexp="\.(php|gif|css|jpg|png)"/>
   <forward regexp="^" target="/index.php"/>
 </rewrite-dispatch>

</web-app>
```

Example: Mediawiki dispatch in resin-web.xml

24.10.30 <rewrite-real-path>

child of: cluster, host, web-app

<rewrite-real-path> lets you rewrite the URL to physical path mapping, to allow aliasing or for filesystem organization.

ATTRIBUTE	DESCRIPTION
real-path	specifies the URL to real-path mapping
regexp	a regular expression matching a URL
replacement	specifies a replacement pattern for URL to URL rewriting
rewrite	rewrites a URL to another URL as a preprocessing-step
target	specifies the target for URL to real-path mapping

<rewrite-real-path> Attributes

```
element rewrite-real-path {
  element rewrite {
    regexp
    & replacement
  }*

  & element rewrite {
    regexp
    & target
  }*
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

  <rewrite-real-path>
    <real-path regexp="^/foo" target="/bar"/>
  </rewrite-real-path>

</web-app>
```

Example: aliasing /foo to /bar

24.10.31 <root-directory>

<root-directory> configures the web-app's filesystem root.

```
element root-directory {
  string
}
```

24.10.32 <secure>

child of: web-app

The <secure> flag requires that the web-app only be accessed in a secure/SSL mode. Equivalent to a <security-constraint>.

```
element secure {
  r_boolean-Type
}
```

24.10.33 <security-constraint>

child of: web-app

Specifies protected areas of the web site. Sites using authentication as an optional personalization feature will typically not use any security constraints. See Resin security for an overview.

Security constraints can also be custom classes.

See Resin security for an overview of security issues and configuration.

ATTRIBUTE	DESCRIPTION
auth-constraint	Defines a security condition based on a logged-in user's role
constraint	Defines a custom security condition
ip-constraint	Defines a security condition based the remote IP address
role-name	Defines role-name requirement

url-pattern	URL pattern to match the security constraint
user-data-constraint	Defines SSL or non-SSL requirements
web-resource-collection	URL patterns and HTTP methods defining the constraint

<security-constraint> Attributes

```
element security-constraint {
  auth-constraint*
  & constraint*
  & ip-constraint*
  & role-name*
  & user-data-constraint*
  & url-pattern?
  & web-resource-collection*
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

<security-constraint>
  <web-resource-collection>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint role-name='user'>
</security-constraint>

</web-app>
```

Example: 'user' role required in WEB-INF/resin-web.xml

24.10.34 <servlet>

Defines a servlet to process HTTP URL requests. The servlet class can either implement javax.servlet.Servlet to handle the HTTP request directly or it can use a remoting protocol like SOAP or Hessian to handle remoting requests.

Since servlets are full Resin-IoC beans, they can use dependency injection, EJB aspects like @TransactionAttribute, custom @InterceptorType interceptors, and listen for @Observes events.

ATTRIBUTE	DESCRIPTION
init	Resin-IoC initialization
init-param	Initialization parameters
load-on-startup	Initializes the servlet when the server starts.
protocol	Protocol configuration for Resin remoting.

run-at	Times to execute the servlet automatically.
servlet-name	The servlet's name (alias)
servlet-class	The servlet's class (In Resin, defaults to servlet-name)

<servlet> Attributes

```
element servlet {
  servlet-name
  < (servlet-class | jsp-file)
  < init*
  < init-param*
  < load-on-startup?
  < protocol?
  < run-as?
  < run-at?
}
```

```
<web-app xmlns="http://caucho.com/ns">

  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>test.HelloWorld</servlet-class>
    <init-param>
      <param-name>title</param-name>
      <param-value>Hello, World</param-value>
    </init-param>
  </servlet>

  <!-- using Resin shortcut syntax -->
  <servlet servlet-name='cron'
           servlet-class='test.DailyChores'>
    <init-param title='Daily Chores'/>
    <load-on-startup/>
    <run-at>3:00</run-at>
  </servlet>

  <!-- mapping a url to use the servlet -->
  <servlet-mapping url-pattern='/hello.html'
                   servlet-name='hello'/>

</web-app>
```

Example: WEB-INF/resin-web.xml

Several `servlet` configurations might configure the same servlet class with different `init-param` values. Each will have a separate `servlet-name` .

```
<web-app xmlns="http://caucho.com/ns/resin">
  <servlet servlet-name='foo-a'>
    <servlet-class>test.FooServlet</servlet-class>
    <init-param name='foo-a sample'/>
  </servlet>

  <servlet servlet-name='foo-b'>
    <servlet-class>test.FooServlet</servlet-class>
    <init-param name='foo-b sample'/>
  </servlet>
</web-app>
```

Example: multiple servlets using the same class

`load-on-startup` can specify an (optional) integer value. If the value is 0 or greater, it indicates an order for servlets to be loaded, servlets with higher numbers get loaded after servlets with lower numbers.

There are a number of named servlets that are usually available to a Resin application, as defined in `$RESIN_HOME/conf/app-default.xml` .

```
  <servlet servlet-name="directory"
           servlet-class="com.caucho.servlets.DirectoryServlet"/>

  <servlet servlet-name="file"
           servlet-class="com.caucho.servlets.FileServlet"/>

  <servlet servlet-name="jsp"
           servlet-class="com.caucho.jsp.JspServlet"/>

  <servlet servlet-name="xtp"
           servlet-class="com.caucho.jsp.XtpServlet"/>

<servlet servlet-name="j_security_check"
         servlet-class="com.caucho.server.security.FormLoginServlet"/>
```

Example: servlet-mappings in $RESIN_HOME/conf/app-default.xml

24.10.35 <servlet-mapping>

Maps url patterns to servlets. `servlet-mapping` has two children, `url-pattern` and `servlet-name` . `url-pattern` selects the urls which should execute the servlet.

ATTRIBUTE	DESCRIPTION
init	Resin-IoC configuration of the servlet.
protocol	Defines the optional remoting protocol.

servlet-class	The servlet-mapping can define the servlet directly as a shortcut.
servlet-name	The servlet name
url-pattern	A pattern matching the url: `/foo/*` , `/foo` , or `*.foo`
url-regexp	A regular expression matching the portion of the url that follows the context path

<servlet-mapping> Attributes

```
element servlet-mapping {
  init?
  & protocol?
  & servlet-class?
  & servlet-name?
  < url-pattern*
  < url-regexp*
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">

  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>test.HelloWorld</servlet-class>
  </servlet>

  <servlet-mapping>
    <url-pattern>/hello.html</servlet-class>
    <servlet-name>hello</servlet-class>
  </servlet-mapping>

  <!-- resin shortcut syntax -->
  <servlet-mapping url-pattern='*.xtp'
                   servlet-name='com.caucho.jsp.XtpServlet'/>

</web-app>
```

Example: WEB-INF/resin-web.xml <servlet-mapping>

`url-regexp` matches the portion of the url that follows the context path. A webapp in `webapps/ROOT` , and a url `http://localhost/foo/hello.html` will have a value of `"/foo/hello.html"` for the purposes of the regular expression match. A webapp in `webapps/baz` and a url `http://localhost/baz/hello.html` will have a url of `"/hello.html"` for the purposes of the regular expression match, because "/baz" is the context path.

In Resin, the special `servlet-name 'invoker'` is used to dispatch servlets by class name. **Warning:** Enabling the `invoker` servlet can create a security hole in your application. Any servlet in the classpath, perhaps even one in a .jar that you are unaware of, could be invoked.

```
<web-app xmlns="http://caucho.com/ns/resin">

  <!--
    used with urls like
    http://localhost:8080/servlets/test.HelloServlet
  -->
  <servlet-mapping url-pattern="/servlet/*" servlet-name="invoker"/>

</web-app>
```

Example: WEB-INF/resin-web.xml servlet invoker

There are a number of mappings to servlets that are usually available to a Resin application, as defined in `$RESIN_HOME/conf/app-default.xml` .

```
<cluster>

<web-app-default>
  <servlet-mapping url-pattern="*.jsp" servlet-name="jsp"/>
  <servlet-mapping url-pattern="*.xtp" servlet-name="xtp"/>

  <servlet-mapping url-pattern="/servlet/*" servlet-name="invoker"/>
  <servlet-mapping url-pattern="/" servlet-name="file"/>
</web-app-default>
</cluster>
```

Example: servlet-mappings in $RESIN_HOME/conf/app-default.xml

The plugins use servlet-mapping to decide which URLs to send to Resin. The following servlet-name values are used by the plugins:

ATTRIBUTE	DESCRIPTION
plugin_match	The plugin will send the request to Resin, but Resin will ignore the entry. Use to get around regexp limitations. (Resin 1.2.2)
plugin_ignore	The plugin will ignore the request. Use this to define a sub-url the web server should handle, not Resin. (Resin 1.2.2)

servlet-name values used by plugins

24.10.36 <servlet-regexp>

Maps URL by regular expressions to custom servlets.

```
element servlet-regexp {
  init?
  & servlet-class?
  & servlet-name?
  & url-regexp
}
```

```
<servlet-regexp url-regexp="/([^.]*).do"
                servlet-class="qa.\${regexp[1]}Servlet">
  <init a="b"/>
</servlet-regexp>
```

24.10.37 <session-config>

<session-config> configures Resin's session handling, including the cookies Resin sends, the maximum sessions, and session persistence and clustering.

See also: Resin clustering for more information about distributed sessions and persistence.

ATTRIBUTE	DESCRIPTION	DEFAULT
always-load-session	Reload data from the store on every request	false
always-save-session	Save session data to the store on every request	false
enable-cookies	Enable cookies for sessions	true
enable-url-rewriting	Enable URL rewriting for sessions	true
cookie-version	Version of the cookie spec for sessions	1.0
cookie-domain	Domain for session cookies	none
cookie-max-age	Max age for persistent session cookies	none
cookie-length	Maximum length of the cookie	
ignore-serialization-errors	When persisting a session, ignore any values which don't implement java.io.Serializable	false
invalidate-after-listener	If true, invalidate the sessions after the session listeners have been called	
reuse-session-id	Reuse the session id even if the session has timed out. A value of false defeats single signon capabilities. (resin 2.0.4)	true
session-max	Maximum active sessions	4096
session-timeout	The session timeout in minutes, 0 means never timeout.	30 minutes

serialization-type	Use one of "java" or "hessian" for serialization, hessian is significantly faster and smaller (resin 3.1.2)	java
save-mode	When to save sessions, one of "before-headers", "after-request", or "on-shutdown"	after-request
use-persistent-store	Uses the current persistent-store to save sessions. (resin 3.0.8)	none

<session-config> Attributes

```
element session-config {
  always-load-session?
  & always-save-session?
  & cookie-append-server-index?
  & cookie-domain?
  & cookie-length?
  & cookie-max-age?
  & cookie-port?
  & cookie-secure?
  & cookie-version?
  & enable-cookies?
  & enable-url-rewriting?
  & ignore-serialization-errors?
  & invalidate-after-listener?
  & reuse-session-id?
  & save-mode?
  & save-on-shutdown?
  & serialization-type?
  & session-max?
  & session-timeout?
  & use-persistent-store?
}
```

The `session-timeout` and `session-max` are usually used together to control the number of sessions. Sessions are stored in an LRU cache. When the number of sessions in the cache fills up past `session-max` , the oldest sessions are recovered. In addition, sessions idle for longer than `session-timeout` are purged.

```
<web-app id='/dir'>

  <session-config>
     <!-- 2 hour timeout -->
     <session-timeout>120</session-timeout>
     <session-max>4096</session-max>
  </session-config>

</web-app>
```

using session-config and session-timeout to control the number of sessions

`cookie-length` is used to limit the maximum length for the session's generated cookie for special situations like WAP devices. Reducing this value reduces the randomness in the cookie and increases the chance of session collisions.

`reuse-session-id` defaults to true so that Resin can share the session id amongst different web-apps.

The class that corresponds to <session-config> is

24.10.38 <shutdown-wait-max>

The maximum time Resin will wait for requests to finish before closing the web-app.**default:** 15s

```
element shutdown-wait-max {
  r_period-Type
}
```

24.10.39 <statistics-enable>

default: false

<statistics-enable> enables more detailed statistics for the `WebAppMXBean` administration. The statistics gathering is disabled by default for performance reasons.

```
element statistics-enable {
  r_boolean-Type
}
```

24.10.40 <strict-mapping>

default: false, allowing /foo/bar.jsp/foo.

Forces servlet-mapping to follow strict Servlet 2.2, disallowing PATH_INFO. Value is `true` or `false` .

```
element strict-mapping {
  r_boolean-Type
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">
  <strict-mapping>true</strict-mapping>
</web-app>
```

Example: enabling strict-mapping in resin-web.xml

24.10.41 <user-data-constraint>

child of: security-constraint

Restricts access to secure transports, such as SSL

ATTRIBUTE	DESCRIPTION
transport-guarantee	Required transport properties. NONE, INTEGRAL, and CONFIDENTIAL are allowed values.

<user-data-constraints> Attributes

```
element user-data-constraint {
  transport-guarantee
}
```

24.10.42 <web-app>

child of: host, web-app

web-app configures a web application.

ATTRIBUTE	DESCRIPTION	DEFAULT
active-wait-time	how long a thread should wait for the web-app to initialize before returning a 503-busy.	15s
archive-path	Specifies the location of the web-app's .war file.	n/a
context-path	Specifies the URL prefix for the web-app.	the `id` value
id	The url prefix selecting this application.	n/a
redeploy-mode	'automatic' or 'manual', see Startup and Redeploy Mode	automatic
redeploy-check-interval	how often to check the .war archive for redeployment	60s
root-directory	The root directory for the application, corresponding to a url of `/id/` . A relative path is relative to the of the containing . Can use regexp replacement variables.	A relative path constricted with the id or the regexp match
startup-mode	'automatic', 'lazy', or 'manual', see Startup and Redeploy Mode	automatic
startup-priority	specifies a priority for web-app startup to force an ordering between webapps	-1
url-regexp	A regexp to select this application.	n/a

When specified by `id` , the application will be initialized on server start. When specified by `url-regexp` , the application will be initialized at the first

request. This means that `load-on-startup` servlets may start later than expected for `url-regexp` applications.

The following example creates a web-app for /apache using the Apache htdocs directory to serve pages.

```
<host id=''>
  <web-app id='/apache' document-directory='/usr/local/apache/htdocs'>

  ...

</host>
```

The following example sets the root web-app to the IIS root directory.

```
  <web-app id='/' document-directory='C:/inetpub/wwwroot'>
```

When the `web-app` is specified with a `url-regexp` , `document-directory` can use replacement variables (`$2`).

In the following, each user gets his or her own independent application using `user` .

```
<host id=''>

  <web-app url-regexp='/~([^/]*)'
           document-directory='/home/$1/public_html'>

    ...

  </web-app>

</host>
```

24.10.43 <web-app-default>

child of: cluster, host, web-app

Establishes the defaults for a .

When initializing a web-app, all the tags in the web-app-defaults sections configure the web-app. In other words, the web-app-default value is essentially a macro that is cut-and-pasted before the web-app configuration.

web-app-default is used for defining server-wide behavior, like *.jsp handling, and for host-wide behavior.

```
<host>
  <web-app-default>
    <servlet servlet-name='test'
             servlet-class='test.MyServlet'/>

    <servlet-mapping url-pattern='*.text' servlet-class='test'/>
  </web-app-default>
</host>
```

24.10.44 <web-app-deploy>

child of: host, web-app

Specifies war expansion.

web-app-deploy can be used in web-apps to define a subdirectory for war expansion. The tutorials in the documentation use web-app-deploy to allow servlet/tutorial/helloworld to be an independent war file.

ATTRIBUTE	DESCRIPTION	DEFAULT
archive-directory	directory containing the .war files	value of `path`
dependency-check-interval	How often to check the .war files for a redeploy	60s
expand-cleanup-fileset	defines the files which should be automatically deleted when an updated .war expands	all files
expand-directory	directory where wars should be expanded	value of `path`
expand-prefix	prefix string to use when creating the expansion directory, e.g. _war_	
expand-suffix	prefix string to use when creating the expansion directory, e.g. .war	
path	The path to the webapps directory	required
redeploy-mode	"automatic" or "manual"	automatic
require-file	additional files to use for dependency checking for auto restart	
startup-mode	"automatic", "lazy" or "manual"	automatic
url-prefix	url-prefix added to all expanded webapps	""
versioning	if true, use the webapp's numeric suffix as a version	false
web-app-default	defaults to be applied to expaned web-apps	
web-app	overriding configuration for specific web-apps	

<web-app-deploy> Attributes

```
element web-app-deploy {
  archive-directory?
  expand-cleanup-fileset?
  expand-directory?
  expand-prefix?
  expand-suffix?
  path?
  redeploy-check-interval?
  redeploy-mode?
  require-file*
  startup-mode?
  url-prefix?
  versioning?
  web-app-default*
  web-app*
}
```

Overriding web-app-deploy configuration

The web-app-deploy can override configuration for an expanded war with a matching <web-app> inside the <web-app-deploy>. The <document-directory> is used to match web-apps.

```
<resin xmlns="http://caucho.com/ns/resin">
<cluster id="">
<host id="">

<web-app-deploy path="webapps">
  <web-app context-path="/wiki"
             document-directory="wiki">
    <context-param database="jdbc/wiki">
  </web-app>
</web-app-deploy>

</host>
</cluster>
</resin>
```

Example: resin.xml overriding web.xml

versioning

The versioning attribute of the <web-app-deploy> tag improves web-app version updates by enabling a graceful update of sessions. The web-apps are named with numeric suffixes, e.g. foo-10, foo-11, etc, and can be browsed as /foo. When a new version of the web-app is deployed, Resin continues to send current session requests to the previous web-app. New sessions go to the new web-app version. So users will not be aware of the application upgrade.

24.10.45 <web-resource-collection>

child of: security-constraint

Specifies a collection of areas of the web site for security purposes. See Resin security for an overview.

ATTRIBUTE	DESCRIPTION
web-resource-name	a name for a web resource collection
description	
url-pattern	url patterns describing the resource
http-method	HTTP methods to be restricted.
method	

<web-resource-collection> Attributes

```
element web-resource-collection {
  url-method*
  & http-method*
  & web-resource-name?
}
```

24.10.46 <welcome-file-list>

child of: web-app

Sets the files to use as when no filename is present in url. According to the spec, each file is in a <welcome-file> element.

```
element welcome-file-list {
  string
  | welcome-file*
}
```

```
<web-app xmlns="http://caucho.com/ns/resin">
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.xtp</welcome-file>
    <welcome-file>home.xtp</welcome-file>
  </welcome-file-list>
</web-app>
```

Example: WEB-INF/resin-web.xml

Resin also provides a shortcut where you can just list the files:

```
<web-app xmlns="http://caucho.com/ns/resin">
  <welcome-file-list>
    index.jsp, index.xtp, home.xtp
  </welcome-file-list>
</web-app>
```

Example: WEB-INF/resin-web.xml

default: in $RESIN_HOME/conf/app-default.xml is index.xtp, index.jsp, index.html.