Introduction to Jetty

07/06 Version 1.2

Introduction to Jetty

This is an introduction to Jetty. This document is divided into two major sections. The first part discusses Jetty with respect to LogicBlaze FUSE and shows some examples of how LogicBlaze FUSE is using Jetty. The second part is a compilation of existing Jetty documents from www.mortbay.org/jetty/tut/, which is included here for your convenience.

We would like to thank the creators of Jetty and the original authors of Jetty documentation for allowing us to modify these documents to better serve you.

Contents

1. Jetty in LogicBlaze FUSE

Jetty is both an HTTP Server (like Apache) and a Servlet Container (like Tomcat) running as a single Java component. It can be used stand-alone to deploy your static content, servlets, JSPs and Web applications. Alternatively, it can be embedded into your Java application to add HTTP and Servlet capabilities.

Jetty is bundled into LogicBlaze FUSE and is used by various LogicBlaze FUSE applications, such as the LogicBlaze FUSE console and the loan broker demonstration.

The LogicBlaze FUSE distribution includes a pre-configured, pre-installed version of Jetty (version 6). For ease-of-use Jetty is started when LogicBlaze FUSE is started, so you don't need to worry about installation, running, and configuration of Jetty. It is ready to use upon LogicBlaze FUSE start-up. Jetty is embedded in the LogicBlaze FUSE code and adds web container capabilities to the distribution.

Deploying your Web applications on the embedded Jetty is similar to deploying on other installations of Jetty. This section uses LogicBlaze FUSE Web applications to illustrate how to deploy Web applications on LogicBlaze FUSE's embedded Jetty. The general steps can be used to deploy any Web application on the bundled Jetty.

1.1. Adding Web Applications to LogicBlaze FUSE

To deploy your Web applications on the LogicBlaze FUSE Jetty platform do the following steps:

1. Inside the top level of your directory structure you will create the root of your application.

   - LogicBlaze FUSE binary download:

     ```
     [fuse_dir]\components\activemq\servicemix\portal\webapps\[YourWebAppNam]
     ```

   - LogicBlaze FUSE source download:

     ```
     [fuse_dir]\target\fuse-1.x-SNAPSHOT\fuse-1.x-
     SNAPSHOT\components\activemq\servicemix\portal\webapps\[YourWebAppName]
     ```

   > The directory pathnames are slightly different for binary and source downloads of LogicBlaze FUSE, so this is shown where applicable.

   Use the following rules for your directory structure:

   - All files that must be visible to the client browser (.html, .jsp, etc) should be placed in your root directory. If your application is large you may use multiple subdirectories.
   - **`/WEB-INF/web.xml`** file is the Web Application Deployment Descriptor for your web application. This XML file describes all the servlets and components that make up your web application. It also contains all the initialization parameters and container managed security constraints that you want the server to enforce. This file defines everything about your application that the server will need to know except for the context path (which is defined by the system administrator on application deployment).

- **/WEB-INF/classes/** file is used for storing Java classes along with any other resources your applications may need. If you choose to organize classes into Java packages it must be done in this directory.
- **/WEB-INF/lib/** directory is used for holding JAR files that contain Java class files (and other resources like third party class libraries and JDBC drivers) needed to run your application.

2. After adding your Web application to the proper directories edit `jetty-xbean.xml` using the code segment below so that Jetty can deploy your Web application. Your code should be added to the section of `jetty-xbean.xml` where similar code is located.

   - Binary Install:

     ```
     [fuse_dir]\components\activemq\servicemix\portal\jetty-xbean.xml
     ```

   - Source Install:

     ```
     [fuse_dir]\target\fuse-1.x-SNAPSHOT\fuse-1.x-
     SNAPSHOT\components\activemq\servicemix\portal\jetty-xbean.xml
     ```

     For example:

     ```
     <webAppContext contextPath="<YourWebAppName>";
             resourceBase="${xbean.current.dir}/webapps/<YourWebAppName>">;
     <serverClasses>
     </webAppContext>
     ```

1.2. Deploying the LogicBlaze FUSE Console Web Client

The LogicBlaze FUSE Console is a Web application and runs in the embedded Jetty container.

1. The `liferay` application can be found in the `webapps` folder.
2. The console is launched using Jetty's XML deployment plan, `jetty-xbean.xml`:

   ```
   <webAppContext contextPath="/liferay"
   resourceBase="${xbean.current.dir}/webapps/liferay">
    <serverClasses />
   </webAppContext>
   ```

3. The configuration for the console Web application is located in the `liferay\WEB-INF\web.xml` file.

For general information on the console please refer to the *LogicBlaze FUSE Console Guide*.

1.3. Deploying the Loan Broker Demonstration

The Loan Broker is an application that finds the lowest loan rate for its customers. The customer interface to the Loan Broker application is a Web client. The loan broker Web client is configured using the same general steps as the LogicBlaze FUSE console and all other Jetty deployments:

1. The loan broker Web client, `loanbroker-client`, can be found in the `webapps` folder.
2. The Jetty deployment descriptor, `jetty-xbean.xml`, deploys the loan broker:

```
<webAppContext contextPath="/loanbroker-client"
resourceBase="${xbean.current.dir}/webapps/loanbroker-client">
<serverClasses />/webAppContext>
```

3. The configuration for the console Web application is located in the `WEB-INF\web.xml` file of the `loanbroker-client` application.

For general information on the loan broker please refer to the *Loan Broker Tutorial*.

2. Configuration: Changing Jetty's Port Number

On starting up LogicBlaze FUSE, if you see a message similar to "address already in use" your host already has a web server installed that is using port 8080.

To resolve this modify Jetty's port number as follows:

1. Stop LogicBlaze FUSE. For shutdown instructions please see the Getting Started with LogicBlaze FUSE manual.
2. Edit the `jetty-xbean.xml` file. The file is located:
   - Binary install:

     ```
     [fuse_dir]\components\activemq\servicemix\portal\jetty-xbean.xml
     ```

   - Source install:

     ```
     [fuse_dir]\target\fuse-1.x-SNAPSHOT\fuse-1.x-
     SNAPSHOT\components\activemq\servicemix\portal\jetty-xbean.xml
     ```

   Search for a line similar to: `<nioConnector port="8080"/>`. Change 8080 to an available port number. The suggested range of port numbers is between 8000 and 9900; 8180, 8280, 8380 are all good choices if they are available on your server.
3. Save and exit the `jetty-xbean.xml` file.
4. Start LogicBlaze FUSE. Please see the Getting Started with LogicBlaze Fuse manual for instructions.

Do not edit anything else in the `jetty-xbean.xml` file, especially the web `contextPath`, as it will impact the ability to run the console.

---

3. Getting Started with the Jetty Package

The remainder of this document is a modified version of the Jetty Tutorial located at www.mortbay.org/jetty/tut/.

3.1. Lightweight or Standards Based

Jetty can be used at two different levels: as the core Http Server and the complete Jetty Server. The former provides an HTTP server with the ability to serve static content and servlets, whilst the latter supports richer configuration capabilities and the deployment of standard web applications.

The advantages of the Http Server is that it is lightweight and embeddable and is highly customizable.

If you need standard Servlet Container support for the development and deployment of web applications, then use the enhanced Server (referred to usually as the Jetty Server). This has the added benefit of better support for configuration via XML. Skip down to an example of a web application deployment here.

3.2. Http Server

The `org.mortbay.http.HttpServer` class provides a core HTTP server that listens on specified ports and accepts and handles requests.

The server is configured by method calls on the Java API . This code example creates a simple server listening on port 8080 and serving static resources (files) from the location `./docroot`:

Code example: Creating a trivial HTTP server

```
HttpServer server = new HttpServer();
SocketListener listener = new SocketListener();
listener.setPort(8080);
server.addListener(listener);

HttpContext context = new HttpContext();
context.setContextPath("/");
context.setResourceBase("./docroot/");
context.addHandler(new ResourceHandler());
server.addContext(context);
server.start();
```

The server is made stand-alone by placing the above code in the body of a main method (the `HttpServer` class itself has a main that can be used as an example). To use the server as a component within an application, include the above lines at an appropriate location within the application code.

The `HttpServer` provides a flexible mechanism for extending the capabilities of the server called `HttpHandlers`. The server selects an appropriate `HttpHandler` to generate a response to an incoming request. The release includes handlers for static content, authentication and a Servlet Container.

Servlets are the standard method for generating dynamic content, however the server can also be extended by writing custom `HttpHandlers` if servlets are insufficient or too heavyweight for your application.

The `org.mortbay.http.HttpServer` class also provides a linkage between a collection of request listeners and collections of request handlers:

Diagram: `HttpServer` relationship model

```
HttpListener -> HttpServer --> HttpContext --> HttpHandler
```

It is the responsibility of an `HttpServer` to accept requests received by an `HttpListener`, and match them to suitable `HttpContext(s)`. It does this by using the host and context path elements from the request. Note that more than one `HttpContext` might match the request, and in this case, all `HttpContexts` are tried in the order in which they were registered with the server until the request is marked as having been handled.

The trivial code snippet from the Introduction to the `HttpServer` can then be represented as:

Diagram: Trivial file server object relationships

```
SocketListener --> HttpServer --> HttpContext --> ResourceHandler
port:8080                        "/"             "./docroot"
```

This depicts a single listener on port 8080 passing requests to a single server, which in turn passes them to a single context with a single handler which returns static content from the directory `./docroot`.

3.3. Http Listener

Implementations of the `org.mortbay.http.HttpListener` interface are added to a HttpServer and act as sources of requests for the server. The `org.mortbay.http.SocketListener` is the main implementation. It listens on a standard TCP/IP port for requests, but there are also listener implementations for SSL, Non blocking IO, testing and others.

Multiple listeners may be used to listen on different ports and on specific IP addresses. This is most frequently used with SSL or with multi-hosting:

Diagram: Multiple listeners with multi-hosting

```
SocketListener -+
port:80        |
               |
JSSEListener   -+> HttpServer --> HttpContext --> ResourceHandler
port:443        |                  "/"             "./docroot"
               |
SocketListener -+
host:1.2.3.4
port: 80
```

Listeners are configured via set methods. Listeners can be created by using the HttpServer as a factory to create a standard type of listener:

Code example: Convenience methods for adding standard listeners

```
HttpServer server=new HttpServer();
HttpListener listener=
    server.addListener(new InetAddrPort("myhost",8080));
```

However, in order to provide detailed configuration, it is more common to create the listener directly and then add it to the `HttpServer`:

Code example: Configuring a listener

```
HttpServer server = new HttpServer();
SocketListener listener = new SocketListener();
listener.setHost("myhost");
listener.setPort(8080);
listener.setMinThreads(5);
listener.setMaxThreads(250);
server.addListener(listener);
```

All `HttpListeners` are responsible for allocating threads to requests, so most implementations are extensions of the `org.mortbay.util.ThreadedServer` or `org.mortbay.util.ThreadPool`. Thus attributes such as min/max threads, min/max idle times etc are also set via the API.

Jetty has several type of `HttpListeners` including:

- `org.mortbay.http.SocketListener` for normal http connections.
- `org.mortbay.http.JsseListener` for SSL https connections using JSSE provider.
- `org.mortbay.http.SunJsseListener` for SSL https connections using Suns JSSE provider.
- `org.mortbay.http.SocketChannelListener` for normal http connections using the `java.nio` library for non-blocking idle connections.
- `org.mortbay.http.ajp.AJP13Listener` for integration with apache, IIS etc.

3.4. Http Context

A `org.mortbay.http.HttpContext` aggregates `org.mortbay.http.HttpHandler` implementations. When a request is passed to a `HttpContext` it tries each of its `HttpHandlers` in turn (in the order in which they were registered) until the request is marked as handled. Note that it is perfectly possible for more than one handler to process the request, but only one handler can mark the request as being finally handled.

In Jetty 3.1 and previous releases, the `HttpContext` class was called `HandlerContext`.

A typical a context might have handlers for security, servlets and static resources:

Diagram: Single context, multiple handlers

```
                                                +-> SecurityHandler
                                                |
SocketListener --> HttpServer --> HttpContext +-> ServletHandler
port: 80                           "/"          |
                                                +-> ResourceHandler
```

All `HttpHandlers` within a single `HttpContext` share the following attributes:

- Initialization parameters
- An optional virtual host name for the context
- A path prefix for the context
- A resource base for loading static resources (files/urls)
- A memory cache of resources (files/urls)
- A `ClassLoader` and set of Java permissions
- A request log
- Statistics
- Error page mappings
- MIME type suffix maps

A single `HttpServer` can have multiple `HttpContexts`. This is typically used to serve several applications from the same port(s) using URL mappings:

Diagram: Multiple contexts with URL mapping

```
SocketListener --> HttpServer +-> HttpContext --> HttpHandler(s)
port:80                       |   path:"/alpha/*"
                              |
                              +-> HttpContext --> HttpHandler(s)
                                  path:"/beta/"
```

Alternatively, different applications can be served from the same port using virtual hosts:

Diagram: Multiple contexts with virtual hosts

```
SocketListener --> HttpServer +-> HttpContext --> HttpHandler(s)
port:80                       |   vhost: www.alpha.com
                              |   path: "/"
                              |
                              +-> HttpContext --> HttpHandler(s)
                                  vhost: www.beta.com
                                  path: "/"
```

If multiple contexts are to be served from the same port, but on different IP addresses, then it is possible to give each context its own `HttpServer`:

Diagram: Multiple servers

```
SocketListener --> HttpServer --> HttpContext --> HttpHandler(s)
host:www.alpha.com               path:"/"
port:80
SocketListener --> HttpServer --> HttpContext --> HttpHandler(s)
host:www.beta.com                path: "/"
port:80
```

`HttpContexts` can be instantiated by the `HttpServer` as part of a call to `addContext()` with context `args`:

Code example: Implied context creation

```
HttpContext context = server.addContext("/mydocs/*");
context.setResourceBase("./docroot/");
```

As `addContext()` will always create a new context instance, it is possible to accidentally create multiple copies of the same context (by calling `addContext()` with the same parameters). To avoid this, you can use the `getContext()` method instead, which will only create a new context if one with the same specification does not already exist:

Code example: Lazy context creation

```
HttpContext context = server.getContext("myhost","/mydocs/*");
context.setResourceBase("./docroot/");
```

The previous example highlights that it is possible specify a virtual host as well as the context path. A single context may be registered with different virtual hosts. Once context configuration becomes complex, it is best to take explicit control over context creation:

Code example: Creating a context with multiple virtual hosts

```
HttpContext context = new HttpContext();
context.setContextPath("/context/*");
context.setVirtualHosts(new String[]{"alpa.com","beta.com"});
context.setResourceBase("./docroot/");
server.addContext(context);
```

Derivations of `HttpServer` may implement the `newHttpContext()` method to change the factory method for creating new contexts. This is used, for example, by the `org.mortbay.jetty.Server` class to return `HttpContext` derivations that have convenience methods for configuring servlets. The `org.mortbay.jetty.servlet.WebApplicationContext` class is a specialization of `HttpContext` that configures the handlers by looking at the standard web application XML files.

3.5. Http Handler

The `org.mortbay.http.HttpHandler` interface represents Jetty's core unit of content generation or manipulation. Implementations of this interface can be used to modify or handle requests. Typically, handlers are arranged in a list, and a request presented to each handler in turn until (at most) one indicates that the request has been handled. This allows handlers to:

- Ignore requests that are not applicable
- Handle requests by populating the response and/or generating content
- Modify the request but allow it to pass onto the next handler(s). Headers and attributes may be modified or an `InputStream` filter added
- Modify the response but allow the request to pass onto the next handler(s). Headers may be modified or `OutputStream` filters added.

The handlers provided with the `org.mortbay.http.handler` package are:

- `ResourceHandler` serves static content from the resource base of the `HttpContext`.
- `SecurityHandler` provides `BASIC` and `FORM` authentication.
- `HTAccessHandler` provides apache `.htaccess` style security.
- `NotFoundHandler` handles unserviced requests.
- `DumpHandler` is a debugging tool that dumps the request and response headers.
- `ForwardHandler` forwards a request to another URL
- `NullHandler` is an abstract base implementation of the interface, used for to derive other handlers.

A `ServletHandler` and `WebApplicationHandler` are provided by the `org.mortbay.jetty.servlet` package and is discussed in detail in the Jetty Server section.

`HttpHandlers` are tried within a context in the order they were added to the `HttpContext`. The following code creates a context that checks authentication, then tries a servlet mapping before trying static content then finally dropping through to an error page generator if no handler marks the request as handled:

Code example: Importance of handler ordering

```
HttpContext context = server.addContext("/");
context.add(new SecurityHandler());
context.add(new ServletHandler());
context.add(new ResourceHandler());
context.add(new NotFoundHandler());
```

3.6. Resource Handler

One of the most common things for a `HttpServer` to do is to serve static content from a base directory or URL. The `org.mortbay.http.handler.ResourceHandler` implementation of `HttpHandler` is provided for this purpose. Its features include:

- Support for `GET`, `PUT`, `MOVE`, `DELETE`, `HEAD` and `OPTIONS` methods.
- Handling of `IfModified` headers.
- HTTP/1.1 Range support for partial content serving.
- Index/welcome files.
- Generation of directory listings.

The root directory or URL for serving static content is the `ResourceBase` of the `HttpContext`. Thus, to serve static content from the directory ./`docroot`/:

Code example: Detailed configuration of a `ResourceHandler`

```
HttpContext context = server.getContext("/context/*");
context.setResourceBase("./docroot/");
ResourceHandler handler = new ResourceHandler();
handler.setDirAllowed(true);
handler.setPutAllowed(false);
handler.setDelAllowed(false);
handler.setAcceptRanges(true);
context.addHandler(handler);
context.addHandler(new NotFoundHandler());
```

The `NotFoundHandler` is added to generate a 404 for requests for resources that don't exist. The `ResourceHandler` lets requests that it cannot handle fall through to the next handler.

`HttpHandlers` ARE ORDER DEPENDANT. If in the above example the `NotFoundHandler` had been added to the context before the `ResourceHandler`, then all requests would be 404'd and resources would not be served. It is a common mistake to put a `ResourceHandler` before a `ServletHandler` with the `JSPServlet`, so jsp source code is served rather than the dynamic content from the `JSPServlet`.

3.7. Putting It All Together

Finally, here is a fully worked code example to configure a server on port 8181 serving static content and a dump servlet at /mystuff/.

Code example: Setting up an `HttpServer`

```
import java.io.*;
import java.net.*;
import org.mortbay.util.*;
import org.mortbay.http.*;
import org.mortbay.jetty.servlet.*;
import org.mortbay.http.handler.*;
import org.mortbay.servlet.*;

public class SimpleServer
{
  public static void main (String[] args)
    throws Exception
  {
    // Create the server
    HttpServer server=new HttpServer();

    // Create a port listener
    SocketListener listener=new SocketListener();
    listener.setPort(8181);
    server.addListener(listener);

    // Create a context
    HttpContext context = new HttpContext();
    context.setContextPath("/mystuff/*");
    server.addContext(context);

    // Create a servlet container
    ServletHandler servlets = new ServletHandler();
    context.addHandler(servlets);

    // Map a servlet onto the container
    servlets.addServlet("Dump","/Dump/*","org.mortbay.servlet.Dump");

    // Serve static content from the context
    String home = System.getProperty("jetty.home",".");
    context.setResourceBase(home+"/demo/webapps/jetty/tut/");
    context.addHandler(new ResourceHandler());

    // Start the http server
    server.start ();
  }
}
```

4. Introduction to the Web Application Server

The `org.mortbay.jetty.Server` class extends `org.mortbay.http.HttpServer` with XML configuration capabilities and a J2EE compliant servlet container.

The following code example demonstrates the creation of a server, listening on port 8080 to deploy a web application located in the directory `./webapps/myapp`:

Code example: Creating a Web Application Server

```
Server server = new Server();
SocketListener listener = new SocketListener();
listener.setPort(8080); server.addListener(listener);
server.addWebApplication("/","./webapps/myapp/");
server.start();
```

As mentioned before, the Jetty Server is able to be configured via XML as an alternative to cutting code. The same web application as coded above can be deployed by this XML configuration file:

XML example: Configuring a Web Application Server

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC
 "-//Mort Bay Consulting//DTD Configure 1.2//EN"
 "http://jetty.mortbay.org/configure_1_2.dtd">
<Configure class="org.mortbay.jetty.Server">
  <Call name="addListener">
    <Arg>
      <New class="org.mortbay.http.SocketListener">
        <Set name"port"<8080</Set>
      </New>
    </Arg>
  </Call>
  <Call name="addWebApplication">
    <Arg>/</Arg>
    <Arg>./webapps/myapp/</Arg>
  </Call>
</Configure>
```

To run Jetty with this XML file, execute this command:

```
java -jar start.jar myserver.xml
```

4.1. Using Servlet Handlers

If you do not wish to use web applications but you want to deploy servlets, then you need to register at least one `context` and at least the `ServletHandler` with the server. You are able to statically configure individual servlets at a specific URL pattern, or use dynamic mapping to extract servlet names from the request URL.

The `ServletHandler` can be used with a `HttpServer`.

Code example: Using `ServletHandler` in `HttpServer`

```
HttpServer server = new HttpServer();
server.addListener(":8080");
HttpContext context = server.getContext("/");
ServletHandler handler= new ServletHandler();
handler.addServlet("Dump","/dump/*",
                   "org.mortbay.servlet.Dump");
context.addHandler(handler);
```

Alternately, the `org.mortbay.jetty.Server` can be used instead of a `HttpServer`, so that it's convenience methods may be used:

Code example: Using `ServletHandler` in Server

```
Server server = new Server();
server.addListener(":8080");
ServletHttpContext context = (ServletHttpContext)
    server.getContext("/");
context.addServlet("Dump","/dump/*",
                   "org.mortbay.servlet.Dump");
```

4.2. Using Static Servlet Mappings

The examples above used defined servlet mappings to map a request URL to a servlet. Prefix (eg. "/dump/*"), suffix (eg." *.jsp"), exact (eg "/path") or default ("/") mappings may be used and they are all within the scope of the context path:

Code example: Static servlet mappings

```
Server server = new Server();
server.addListener(":8080");
ServletHttpContext context = (ServletHttpContext)
    server.getContext("/context");
context.addServlet("Dump&quot;,"/dump/*",
                   "org.mortbay.servlet.Dump");
context.addServlet("Dump&quot;,"/dump/session",
                   "org.mortbay.servlet.SessionDump");
context.addServlet("JSP&quot;,"*.jsp",
                   "org.apache.jasper.servlet.JspServlet");
context.addServlet(";Default","/",
                   "org.mortbay.jetty.servlet.Default");
Examples of URLs that will be mapped to these servlets are:
/context/dump Dump Servlet by prefix
/context/dump/info Dump Servlet by prefix
/context/dump/session SessionDump Servlet by exact
/context/welcome.jsp JSP Servlet by suffix
/context/dump/other.jsp Dump Servlet by prefix
/context/anythingelse Default Servlet
/anythingelse Not this context
```

4.3. Using Dynamic Servlets

Servlets can be discovered dynamically by using the `org.mortbay.jetty.servlet.Invoker` servlet. This servlet uses the request URI to determine a servlet class or the name of a previously registered servlet:

Code example: Dynamic servlet mappings

```
Server server = new Server();
server.addListener(":8080");
ServletHttpContext context = (ServletHttpContext)
    server.getContext("/context");
context.addServlet("Dump","/dump/*",
                   "org.mortbay.servlet.Dump");
context.addServlet("Invoker&amp;quot;,"/servlet/*",
                  "org.mortbay.jetty.servlet.Invoker");
```

Examples of URLs that will be mapped to these servlets are:

- `/servlet/Dump` Dump servlet by name
- `/servlet/com.acme.MyServlet/info` servlet by dynamic class
- `/servlet/com.mortbay.servlet.Dump` Dump servlet by class or ERROR

By default, the Invoker will only load servlets from the context classloader, so the last URL above will result in an error. The Invoker can be configured to allow any servlet to be run, but this can be a security issue

5. Deploying Web Applications

The Servlet Specification details a standard layout for web applications. If your content is packaged according to these specifications, then simply call the `addWebApplication`(...) methods on the `org.mortbay.jetty.Server` instance, specifying at minimum a context path, the directory or war file of your application. Jetty is then able to discover and configure all the required handlers including security, static content and servlets.

The `addWebApplication(...)` methods transparently create and return an instance of `WebApplicationContext` which contains a `WebApplicationHandler`.

The `WebApplicationHandler` extends `ServletHandler` and as well as servlets, it provides standard security and filters. Normally it is configured by the `webdefault.xml` file to contain Invoker, JSP and Default servlets. Filters, servlets and other mechanisms are configured from the `WEB-INF/web.xml` file within the web application.

This example configures a web application located in the directory `./webapps/myapp/` at the context path / for a virtual host `myhost`:

Code example: Configuring a web application

```
{{server.addWebApplication("myhost","/","./webapps/myapp/");}}
```

The arguments to the `addWebApplication` method are:

- An (optional) virtual host name for the context
- A context path

The location of the web application, which may be a directory structure or a war file, given as a URL, war filename or a directory name.

The `addWebApplication` method is overloaded to accommodate the parameters marked as (optional).

5.1. Multiple Web Applications

To make things even easier, if you have multiple web apps to deploy, you can accomplish this with a single method call:

Code example: Configuring multiple web apps

```
{{server.addWebApplications ("myhost","./webapps/");}}
```

Given the code above, Jetty would look in the directory `./webapps/` for all war files and subdirectories, and configure itself with each web application specified therein. For example, assuming the directory `webapps` contained the war files `webapps/root.war`, `webapps/customer.war` and `webapps/admin.war`, then Jetty would create the contexts "/", `/customer/* and /admin` mapped to the respective war files.

The special mapping of war files (or directories) named root to the context /.

In order to actually deploy the web application, it is also necessary to configure a port listener. The full code example to deploy the web application in the code snippet is:

Code example: Deploying a web application

```
Server server = new Server();
SocketListener listener = new SocketListener();
listener.setPort(8080);
server.addListener(listener );
server.addWebApplication("myhost","./webapps/myapp/");
server.start();
```

5.2. Using XML

The same web application can be deployed instead via an XML configuration file instead of calls to the API. The name of the file is passed to Jetty as an argument on the command line (see the section on Jetty demonstrations for instructions). The following excerpt deploys the same web application as given in the code example above:

XML example: Deploying a web application

```
<Configure class="org.mortbay.jetty.Server">
  <Call name='addListener">
    <Arg>
      <New class='org.mortbay.http.SocketListener">
          <Set name="Port">
            <SystemProperty name="jetty.port"
            default="8080"/>
          </Set>
      </New>
    </Arg>
  </Call>
  <Call name="addWebApplication">
    <Arg>/</Arg>
    <Arg><SystemProperty name="jetty.home"
          default="."/>/webapps/myapp
    </Arg>
  </Call>
 </Configure>
```

An explanation of the Jetty XML syntax can be found in the section on Jetty XML Configuration.

5.3. Web Application Configuration

When a `WebApplicationContext` is started, up to three configuration files are applied as follows: `webdefault.xml` This file must be in standard `web.xml` format and typically contains all the default settings for all `webapplications`.

By default the `org/mortbay/jetty/servlet/webdefault.xml` file is used as a resource from the jetty jar and it configures the `JspServlet` and default session timeouts. The default xml file may be changed for a particular context by calling `setDefaultsDescriptor(String) web.xml`.

The standard web application configuration file and is found in the `WEB-INF` directory of the Web application. `web-jetty.xml` This file must be in `org.mortbay.xml.XmlConfiguration` format and if found in the `WEB-INFdirectory` of a web application, it is applied to the `WebApplicationContext` instance. It is typically used to change non standard configuration. Note: the name `jetty-web.xml` is also accepted for this file.

XML example: A `web-jetty.xml` file

```
<Configure class="org.mortbay.jetty.servlet.WebApplicationContext">
    <Set name="statsOn" type="boolean"<false>/Set>
</Configure>
```

6. Web Application Security

Jetty makes the following interpretations for the configuration of security constraints within a `web.xml` file:

- Methods `PUT`, `DELETE` and `GET` are disabled unless explicitly enabled.
- If multiple security-constraints are defined, the most specific applies to a request.
- A security-constraint an empty `auth`-constraint forbids all access by any user:

```
<security-constraint>
  <web-resource-collection>
   <web-resource-name>Forbidden</web-resource-name>
    <url-pattern>/auth/noaccess/*</url-pattern>
  </web-resource-collection>
  <auth-constraint/>
</security-constraint>
```

- A security constraint with an `auth`-constraint with a role of * gives access to all authenticated users:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Any User</web-resource-name>
    <url-pattern>/auth/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>*</role-name>
  </auth-constraint>
</security-constraint>
```

- A `security-constraint` with no `auth`-constraint and no data constraint gives access to any request:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Relax</web-resource-name>
    <url-pattern>/auth/relax/*</url-pattern>
  </web-resource-collection>
</security-constraint>
```

On platforms without the / file separator or when the system parameter `org.mortbay.util.FileResource.checkAliases` is true, then the `FileResouce` class compares the `absolutePath` and `canonicalPath` and treats the resource as not found if they do not match. THIS means that win32 platforms need to exactly match the case of drive letters and filenames.

- Dynamic servlets by default, can only be loaded from the context classpath. Use `ServletHandler.setServeDynamicSystemServlets` to control this behavior.

6.1. Security Recommendation

It is strongly recommended that secure `WebApplications` take following approach. All access should be denied by default with

```
    <security-constraint>
      <web-resource-collection>
       <web-resource-name>Default</web-resource-name>
        <url-pattern>/</url-pattern>
      </web-resource-collection>
      <auth-constraint/>
    </security-constraint>
Specific access should then be granted with constraints like:
  <security-constraint>
    <web-resource-collection>
      <url-pattern>/public/*</url-pattern>
      <url-pattern>/images/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>HEAD</http-method>
    </web-resource-collection>
    <web-resource-collection>
      <url-pattern>/servlet/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>HEAD</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>*</role-name>
    </auth-constraint>
  </security-constraint>
```

6.2. Session Security

Jetty uses the standard `java.util.Random` class to generate session IDs. This may be insufficient for high security sites. The `SessionManager` instance can be initialized with a more secure random number generator, such as `java.security.SecureRandom`.

The Jetty configuration XML to do this to a Web application is:

```
<Call name="addWebApplication">
  <Arg>/myapp/*</Arg>
  <Arg><SystemProperty name="jetty.home"
        default="."</demo/webapps/myapp<Arg>
  <Call name="getServletHandler">
    <Set name="sessionManager">
      <New class="org.mortbay.jetty.servlet.HashSessionManager">
        <Arg><New class="java.security.SecureRandom"><Arg>
      </New>
    </Set>
  </Call>
</Call>
```

Initializing the `SecureRandom` object is a one-off time consuming operation which may cause the initial request to take much longer.

7. Additional Resources

http://www.theserverside.com/news/thread.tss?thread_id=36594

http://www.sitepoint.com/blogs/2005/09/18/jetty-60-to-provide-new-architecture-for-ajax-apps/