Emerging Technologies for the Enterprise

March 28-29, 2007 Drexel University, Philadelphia, PA

Lightweight Applications with Spring, Hibernate, and Java EE 5

Aaron Mulder
CTO, Chariot Solutions
ammulder@chariotsolutions.com

Presentation Outline

- Introduction
  - The geeky sample application
  - Technology Overview
- Review of each framework
- Conclusion

About the Speaker

- CTO of Chariot Solutions
  - Architecture mentor including Spring, Spring MVC/Web Flow, SOA, etc.
  - Co-developer of the famous sample application
- Open source contributor
  - Committer on Apache Geronimo, ActiveMQ, ServiceMix, OpenEJB, etc.
  - Previous contributor to JBoss, PostgreSQL, etc.

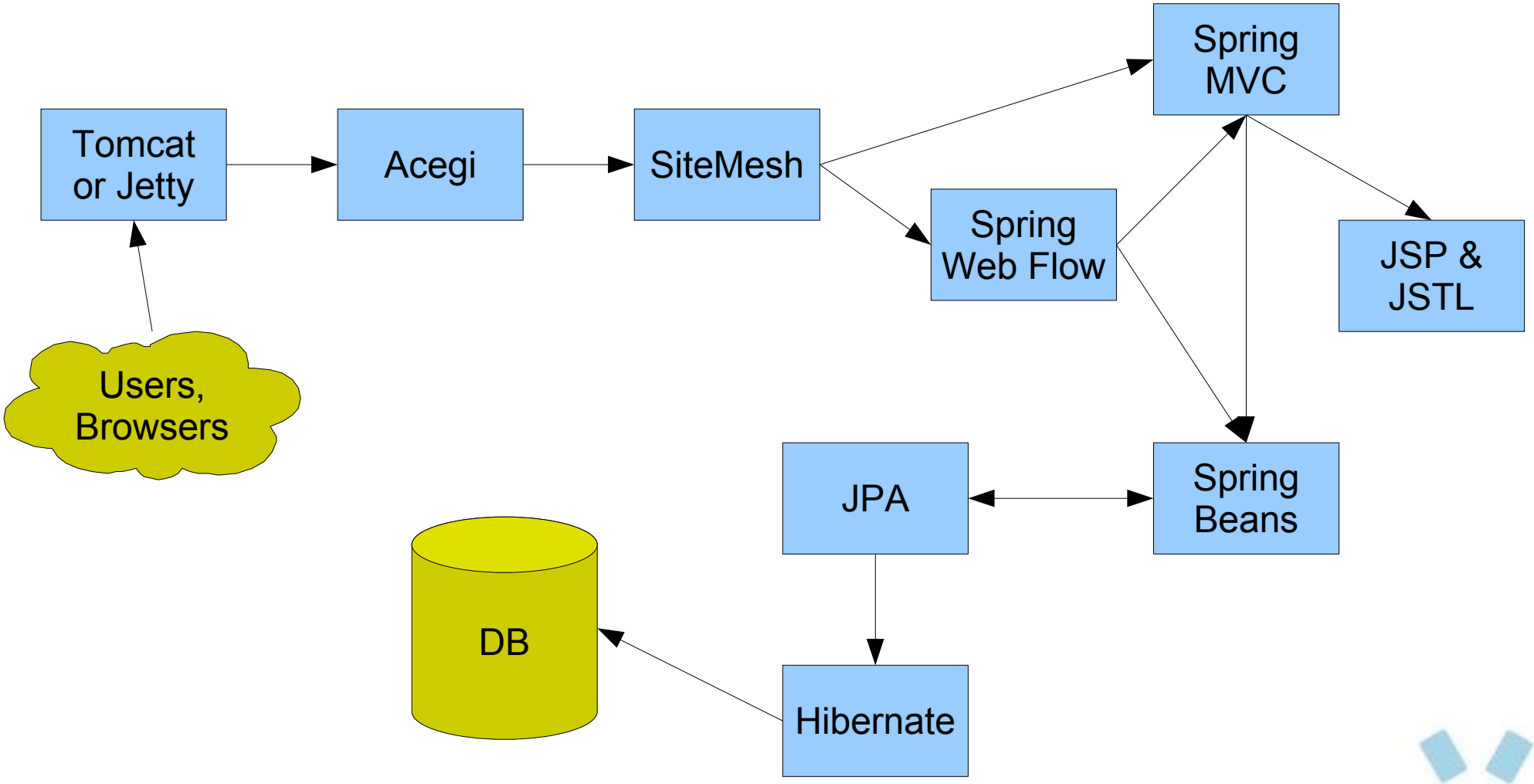
Geeky Sample Application

- I have the same old e-Commerce sample as everybody else
- But...
- I also have an application that was really used this year to manage a convention...
- It was a role-playing games convention, with players and game masters (GMs) and characters (PCs) and game sessions (expeditions) and magic items and so on...
- Has two advantages:
  - 1: Really used in production
  - 2: I'm still working on #2
- Anyway, it beats e-Commerce

Technology Stack

- Java SE 5 (annotations)
- Java EE 5 (JPA, JSP/JSTL)
- Spring (configuration, transactions)
- Spring MVC, Spring Web Flow (web frameworks)
- SiteMesh (page framing / templates)
- Acegi (security)
- Hibernate (JPA implementation)
- Tomcat 6.x or Jetty 6.x ("app server")
- Maven (build)
- HSQLDB (test database)

Technology Diagram



Wow? That Many?!?

- Could eliminate some by going with full Java EE app server
- Still need a web framework, and most of the other options are then limited to one implementation, which is not necessarily better
- This stack can run on any Java EE 5 web container or better, plus any web hosting service with a modern web container
- The middle-tier code is POJOs with annotations, and you could easily add EJB annotations to run as EJBs instead of in Spring; JPA entities can be run in the container already
- Web container, SiteMesh, Hibernate, Maven, and Acegi are essentially transparent once configured the first time

Technology Walkthrough

Java SE 5 Annotations

- Replaces XML, XML, XML (often complex/ugly XML)
- Compiler/IDE will enforce more syntax
- Easier to tell at a glance what settings apply to what objects or methods
- Sometimes harder to set up cross-cutting effects (e.g. apply transactions to set*)
- Overall, seems beneficial

Before Annotations

- Spring Transactions: old style

```
<!-- the transactional advice (i.e. what 'happens';
     see the <aop:advisor/> bean below) -->
<tx:advice id="txAdvice" transaction-manager="txManager">
  <!-- the transactional semantics... -->
  <tx:attributes>
    <!-- all methods starting with 'get' are read-only -->
    <tx:method name="get*" read-only="true"/>
    <!-- other methods use the default transaction settings (see below) -->
    <tx:method name="*"/>
  </tx:attributes>
</tx:advice>

<!-- ensure that the above transactional advice runs for any execution
     of an operation defined by the FooService interface -->
<aop:config>
  <aop:pointcut id="playerMgrOperation"
                expression="execution(* x.y.PlayerManagerBean.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="playerMgrOperation"/>
</aop:config>
```

With Annotations

- Spring Transactions: with annotations

```
@Transactional(readOnly = true, rollbackFor = ConException.class)
public class PlayerManagerBean implements PlayerManager {

    public Player getPlayerByUsername(int eventID,
                                      String username) {
        ...
    }

    @Transactional(readOnly = false,
                   rollbackFor = ConException.class)
    public Player createPlayer(Player p) throws ConException {
        ...
    }
```

Annotations – worth it?

- I would say so
  - Less verbose
  - Keeps you focused on code
  - Not lost in morass of XML
  - Easy to notice if omitted from a class/method
  - Automatically picked up at runtime, or ignored if run in an environment that doesn't use them
  - IntelliJ has good error checking, argument information, and code completion for annotations

Java Persistence API (JPA)

- Last year, would have just used Hibernate
- Now, for little or no loss of features, can be portable to many other persistence providers *and* EJB
  - Well, JPA doesn't have an API for constructing queries programmatically, but does have a very powerful QL to construct them as Strings
- Based on annotations & objects are POJOs
  - Very simple class structure
  - JPA entity objects are often reused directly by higher tiers, so they may end up with some helper methods to reformat data into text, etc.

Some JPA Disadvantages

- Why would you not use JPA?
  - 1 annoyance: pre-configured queries and named parameters are identified only be name -- compiler can't catch errors. Typos are discovered at runtime.
    - But you aren't required to pre-configure queries, can just call a query method and pass it JPA QL or SQL
  - 2 annoyance: navigating relationships only works “during the session", so the middle tier needs to pre-access lazy relationships that the front end will want to use
    - getAllX vs. getX(id) vs getFullyLoadedX(id)
    - Not unique to JPA
    - Can also arrange for session to include front-end execution, but that has other effects...

JPA Entity Example

```
@Entity
@SequenceGenerator(name = "PlayerSequence", sequenceName = "PLAYER_SEQUENCE")
@NamedQueries({
        @NamedQuery(name = "Player.findForEvent",
                query = "select object(p) from Player p join p.roles as r " +
                        "where r.eventId = :eventID and r.role='Player' "+
                        "order by p.firstName, p.lastName")
})
public class Player implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
    private int id;                                          "PlayerSequence")
    private String username;
    private String password;
    @Column(name = "first_name")
    private String firstName;
    ...
    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    public String getUsername() {return username;}
    public void setUsername(String username) {this.username=username;}
    ...
}
```


JPA Queries

```
@NamedQueries({
    @NamedQuery(name = "Player.findForEvent",
        query = "select p from Player p join p.roles as r " +
            "where r.eventId = :eventID and r.role='Player' "+
            "order by p.firstName, p.lastName")
})
```

```
private EntityManager em;
...
public List<Player> getAllPlayers(int eventID) {
    Query qry = em.createNamedQuery("Player.findForEvent");
    qry.setParameter("eventID", eventID);
    return qry.getResultList();
}
```

JPA Relationships

```
public class Player implements Serializable {
    ...
    @OneToMany(mappedBy = "player")
    private Collection<PC> pcs;

    public Collection<PC> getPCs() {return pcs;}
```

```
public class PC implements Serializable {
    ...
    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "player_id", updatable = false)
    private Player player;

    public Player getPlayer() {return player;}
```

JPA Sorted Relationships

- A collection-type relationship is normally ordered by primary key, but you can override that:

```
public class Player implements Serializable {
    ...
    @OneToMany(mappedBy = "player")
    @OrderBy("name asc")
    private Collection<PC> pcs;
```

JPA Sessions & Relationships

- JPA objects are fully "live" when operating within a JPA session (normally, a transaction). Once they leave the transaction (e.g. value returned to front end), you can't load new data (navigate new relationships). So... may want back end to pre-navigate/pre-load...

```
Query query = ...
List<Player> list = query.getResultList();
for (Player player : list) {
    if(player.getCurrentPC() != null)
        player.getCurrentPC().
                        getCurrentExpedition();
}
return list;
```


JPA Inserts & Complex Logic

- When first inserting, relationships tend to be null, all objects lack IDs, etc. If you're than going to hand off to reused business logic, you may want to refresh the instance so it's all in order...

```
pc.setSomething(...);
pc.setSomethingElse(...);
em.persist(pc);          // Save the main instance
initializeSkills(pc); // Save related stuff
initializeFeats(pc);  // Save more related stuff
em.flush();
em.refresh(pc);
recalculate(pc);         // Expects a working PC
return pc;
```

Using Hibernate via JPA

- All the code uses JPA APIs. Just need some configuration to say "use Hibernate as the JPA implementation". This can go in the JPA or Spring config files. Here it is for JPA:

```
<persistence-unit name="MyApplication">
<provider>o.h.ejb.HibernatePersistence</provider>
  <properties>
    <property name="hibernate.show_sql"
              value="true"/>
    <property name="hibernate.dialect"
          value="o.h.dialect.PostgreSQLDialect"/>
  </properties>
</provider>
</persistence-unit>
```

JPA – worth it?

- Huge advance over old options; comparable to straight Hibernate, etc.
  - Often contrasted to using Hibernate directly
  - The standard has close to all of the product-specific features, so why not give yourself that portability?
  - Supported by other tools/frameworks
  - Trivial to port to a full Java EE / EJB application
  - Seems worthwhile to me

Spring

- Glue code and configuration/annotations for:
  - Database Pool & JPA
  - Transactions
  - Business Logic
  - Security
  - Web tier
- Dependency Injection
  - Wiring by injection rather than by JNDI lookup

Spring Very Basics

- All configuration done in XML files
- Can split across several XML files
- Config file names and configures bean instances
- Assign beans to properties of other beans
- Spring will call the setters before the beans are used, so all needed references are present and valid
- Many built-in APIs and many extensions to integrate various tools and libraries into Spring

Spring in the Sample Application

- Set up 3 config files
  - One for business logic and middle-tier services
  - One for web application components
  - One for security
- Lots of boilerplate or initial configuration
  - Then just add an entry for each new service or URL handler
- Much of the nasty XML from Spring 1 & Java < 5 turns into pleasing annotations now

A Spring Business Service: Header

- The business logic beans are POJOs with annotations:

```
@Repository
@Transactional(readOnly = true,
               rollbackFor = ConException.class)
public class PCManagerBean implements PCManager {
    private EntityManager em;

    @PersistenceContext
    public void setEntityMgr(EntityManager em) {
        this.em = em;
    }

    ...
```

A Spring Business Service: Methods

- Business methods are straightforward

```
public class PCManagerBean implements PCManager {
    private EntityManager em;
    ...
    public List<Player> getPlayersWithoutPCs(
                                    int eventID) {...}

    @Transactional(readOnly = false,
                   rollbackFor = ConException.class)
    public GeneratePC startNewPC(int playerID,
           int eventID) throws ConException {...}
    ...
}
```

Spring for Business Logic

- Those Spring business logic beans have:
  - Optional interface with the implementation
  - Transaction settings (read/write, requires/requires new, exceptions to rollback for) via annotations
  - Translation from DB-specific exceptions to Spring standard exceptions via an annotation
  - Acegi Security via annotations
  - JPA EntityManager injected via an annotation
  - References to other services injected (though configured in XML)

More on Spring Transactions

- Can use annotations at the bean or method level
- Method annotations take priority and cause all bean-level TX annotations to be ignored for that method
  - Method statements should include all the settings from the bean level; don't assume omitted ones will be "inherited" from bean-level annotations
- Checked exceptions do not cause rollback by default; Runtime exceptions do.
  - Can list checked exception classes that should cause rollback in the annotations

Business Logic Config: Persistence

- A regular Spring config file: this is part 1/3

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
                                                    destroy-method="close">
    <property name="driverClassName" value="org.postgresql.Driver"/>
    <property name="url" value="jdbc:postgresql://localhost/mydatabase"/>
    <property name="username" value="scott"/>
    <property name="password" value="tiger"/>
</bean>

<bean id="entityManagerFactory"
              class="o.s.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="myDS"/>
</bean>

<bean id="transactionManager" class="o.s.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```

Business Logic Config: Annotations

- Part 2/3 deals with annotations:

```
<!-- Enable the configuration of transactional
     behavior based on annotations -->
<tx:annotation-driven/>

<!-- JPA annotations bean post processor -->
<bean class="o.s.orm.jpa.support.
          PersistenceAnnotationBeanPostProcessor"/>

<!-- Exception translation bean post processor
    (based on Repository annotation) -->
<bean class="o.s.dao.annotation.
    PersistenceExceptionTranslationPostProcessor"/>
```

Business Logic Config: The Beans

- Part 3 has the actual beans; the interesting part (but note, no JPA, TX, security, etc.)

```
<bean name="dice"
      class="org.princecon.logic.beans.DiceBean" />

<bean name="itemManager"
  class="org.princecon.logic.beans.ItemManagerBean" />

<bean name="playerManager"
  class="org.princecon.logic.beans.PlayerManagerBean">
      <property name="dice" ref="dice" />
</bean>
```

Spring – worth it?

- Provides many of the expected "enterprise features" in a lightweight environment
  - Security, transactions, etc.
- Nearly automatically links business logic to JPA entities and the database
- As we'll see, also links web controllers to business logic
- To summarize: How could you write a lightweight application without it?

Spring MVC Web Framework

- A MVC web framework, similar at a high level to Struts (in contrast to JSF/Tapestry)
  - A "front controller" servlet receives requests and then dispatches them to URL-specific handlers
- The "model" is a map in which any bean or value can be placed
- The "view" can use many view technologies (JSP, Velocity, Freemarker, PDF writers, etc., etc., etc.)
- The "controller" can implement an interface or extend one of several base classes (each best for a different purpose)

Typical Spring MVC

- Usually, not all screens use the same controller base classes
- Little or no code required for simple screens; form handler screens may need more extensive subclasses
- Different controllers have different levels of support for binding request attributes, validation, etc.
- A JSP tag library is included for form elements that should auto-populate to/from a backing bean
  - Handles form and field level validation & errors
  - New and still a bit fragile (in terms of handling incorrect usage)

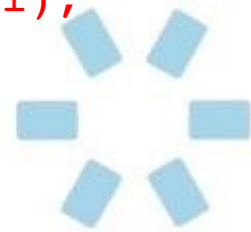
Spring MVC: Simple Controller

- This is a basic controller for a "view" screen

```
public class ViewPlayerController extends AbstractController {
    private PlayerManager mgr;

    // Called by Spring dependency injection
    public void setPlayerManager(PlayerManager mgr) {
        this.mgr = mgr;
    }

    // Handles requests
    protected ModelAndView handleRequestInternal(HttpServletRequest
            request, HttpServletResponse response) throws Exception {
        Integer plyrID = Integer.valueOf(request.getParameter("id"));
        Player player = mgr.getFullyLoadedPlayer(plyrID);
        Map<String, Object> model = new HashMap<String, Object>();
        model.put("player", player);
        return new ModelAndView("staff/player/playerView", model);
    }
}
```

Spring MVC: Simple Controller's JSP

- And here is a corresponding JSP:

```
<%@ taglib prefix="c"
           uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <head>
    <title>
      View Player
            ${player.firstName} ${player.lastName}
    </title>
  </head>
  <body>
    ...
  </body>
</title>
```

Spring MVC: Form Controller

- The form controller has much more logic built in

```
public class CreatePlayerController extends SimpleFormController {
    private PlayerManager mgr;
    ...

    // Handles requests
    protected void doSubmitAction(Object commandObject) {
        Player p = (Player) commandObject;
        PlayerRole pr = ...
        if(p.getId() < 1) {
            Set<PlayerRole> set = Collections.singleton(pr);
            p.setRoles(set);
            p = mgr.createPlayer(p);
        } else {
            p = mgr.updatePlayer(p, pr);
        }
    }
}
```

Spring MVC: Form Controller's JSP

- And here are snippets from the corresponding JSP:

```
<%@ taglib prefix="mvc"
    uri="http://www.springframework.org/tags/form" %>
<mvc:form>
  <mvc:errors cssClass="errors"/> // form-level errors
  <b>First Name:</b>
     <mvc:input path="firstName" size="15"/>
     <mvc:errors path="firstName" cssClass="errors"/>
  <b>Password:</b>
     <mvc:password path="password" size="15"/>
     <mvc:errors path="password" cssClass="errors"/>
  <input type="submit" value="Create Player" />
</mvc:form>
```

Spring MVC Config File

- There are some boilerplate beans to set up how controllers are mapped to URLs (by bean name) and how view names are mapped to JSPs (by adding `/WEB-INF/jsp/` to the beginning and `.jsp` to the end)

```
<bean id="handlerMapping"
class="o.s.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<bean id="viewResolver"
      class="o.s.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass"
  value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

Spring MVC Controllers

- The rest of the Spring MVC config file has validators and controllers:

```
<bean id="playerValidator" class="...PlayerValidator">
    <property name="playerManager" ref="playerManager" />
</bean>

<bean name="player/view.do" class="...ViewPlayerController">
    <property name="playerManager" ref="playerManager" />
</bean>

<bean name="player/add.do" class="...CreatePlayerController">
  <property name="playerManager" ref="playerManager" />
  <property name="commandClass" value="org.princecon.jpa.Player"/>
  <property name="formView" value="player/create" />
  <property name="successView" value="player/playerCreated" />
  <property name="validator" ref="playerValidator" />
</bean>
```

Spring MVC – worth it?

- Great if you're coming from Struts, or are unsold on component-based web frameworks
- Tag library and SimpleFormController helps with form processing
- Once set up, incremental configuration for each screen/controller is pretty minimal
  - But still in XML
- All code is light, but screens with complex behavior still end up with a fair amount of code
- Certainly seems like a reasonable option

Spring Web Flow

- Handles "flows", "business processes", or "wizards" that span several screens
- Sits on top of another web framework that handles some of the infrastructure (view technology, etc.)
  - Supports Spring MVC, Struts, JSF, etc.
- Configured with an XML file for each flow, which is a pretty straightforward mapping of a flow diagram to XML (start states, end states, subflows, view states, action states, decision states)

Spring Web Flow

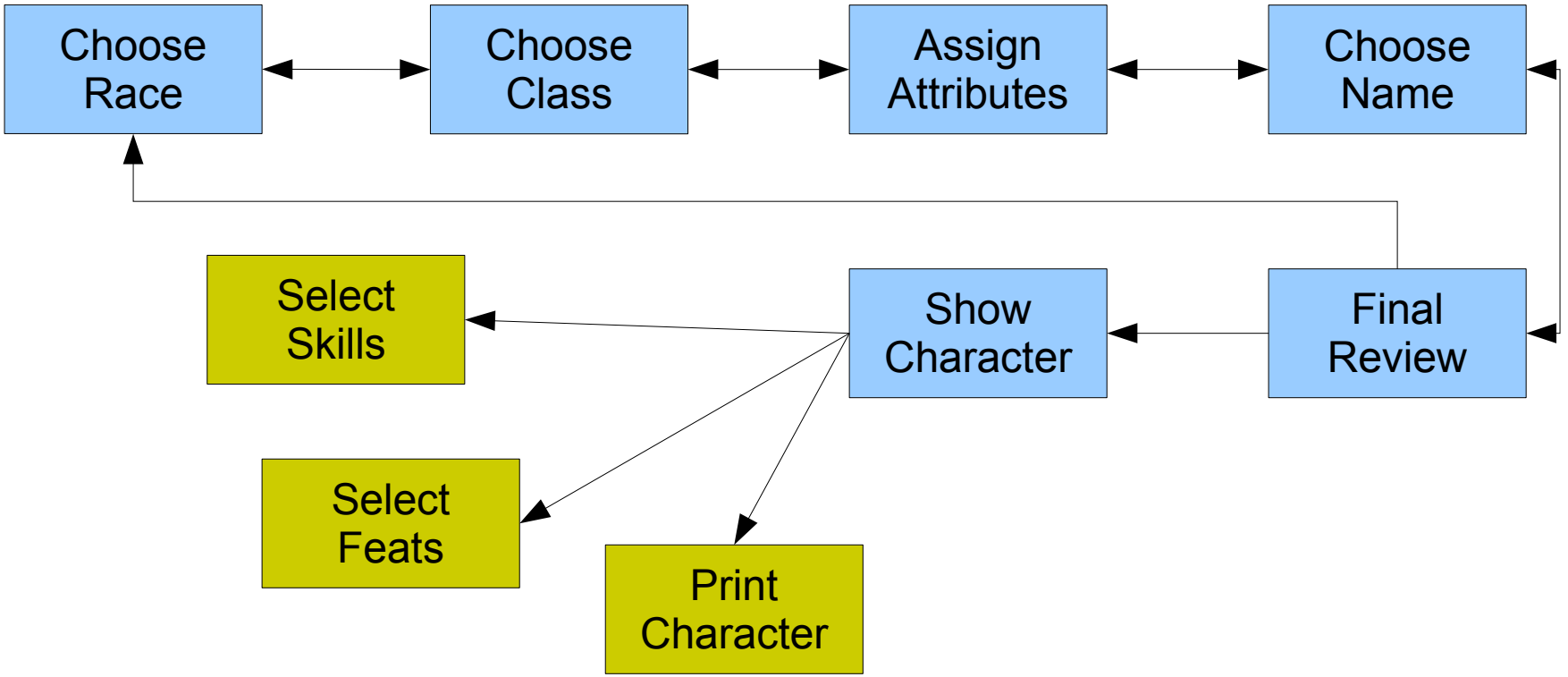
- Special handling for "flows" consisting of several screens the user will go through more or less in order
  - Sign up, Checkout, Wizards, "a process"
- Sits on top of your regular web framework
- Stores progress/data in a separate memory space from the application, with scopes like the flow, the conversation (a flow and any subflows), etc.
- Uses continuations by default to support the back and reload buttons (letting you pick up from any point in your history)
  - At the cost of more memory overhead

Spring Web Flow in Practice

- Add a URL handler for one or many flows
- Request parameters will identify the flow to execute (or the flow in progress)
- Define the flows, each in a separate XML file
- For each flow, define entry and exit points, actions, views, subflows, and the navigation between
- Figure out how the data accumulated in the flow will be represented, and adjust the flow definition or controller methods to access it accordingly

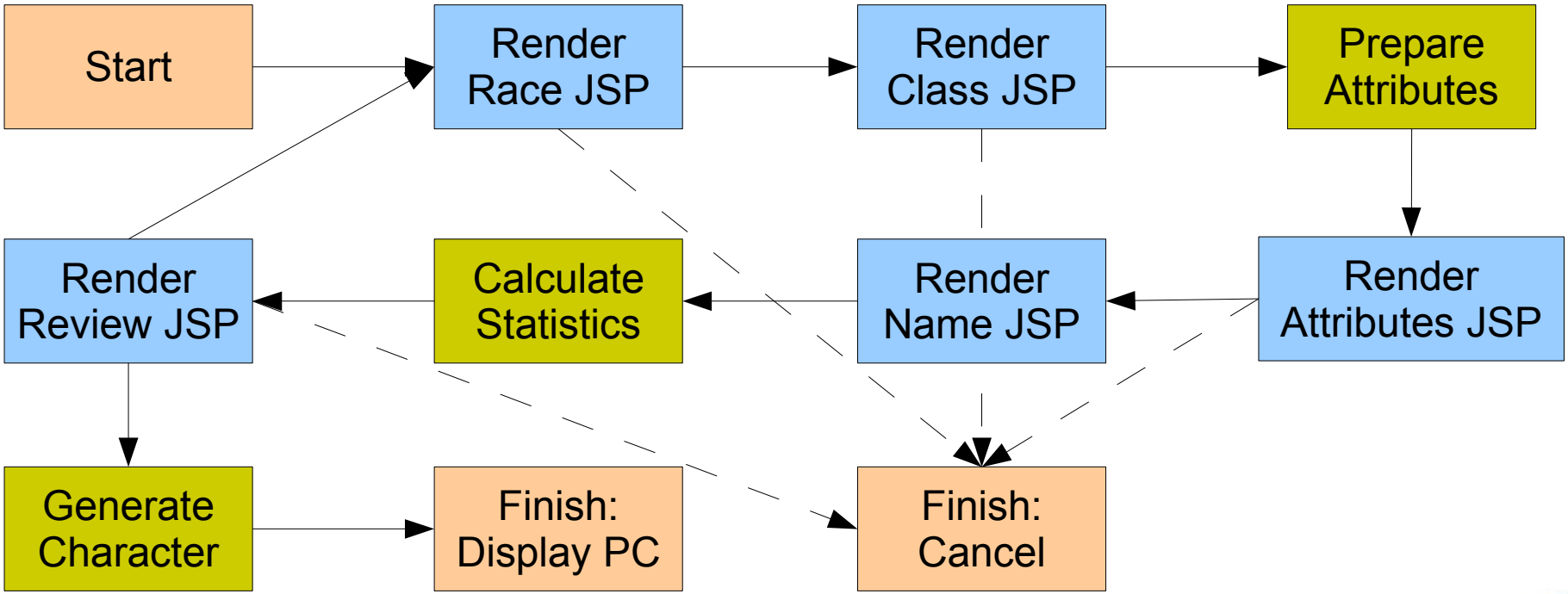
Spring Web Flow Example

- In the sample application, generating a character is a flow, but some character-related screens are not:



Spring Web Flow Example Detail

- In the sample application, generating a character is a flow, but some character-related screens are not:



Spring Web Flow: Flow Definition

- Excerpts from the flow definition XML file:

```
<start-state idref="chooseRace" />

<view-state id="chooseRace" view="createpc/race">
  <render-actions>
    <action bean="chooseRaceController" method="setupForm" />
  </render-actions>
  <transition on="submit" to="chooseClass">
    <action bean="chooseClassController" method="bindAndValidate"/>
  </transition>
  <transition on="cancel" to="cancelCreatePC" />
</view-state>

<action-state id="prepareAttributes">
  <action bean="attributesController" method="prepare" />
  <transition on="success" to="identifyEmployee" />
</action-state>
```


Spring Web Flow: Beans

- The web flow references controllers defined in a Spring config file:

```
<bean id="chooseRaceController"
        class="org.springframework.webflow.action.FormAction">
    <property name="formObjectClass" value="...GeneratePC" />
    <property name="formObjectScope" value="FLOW" />
    <property name="formObjectName" value="pc" />
</bean>

<bean id="attributesController" class="...AttributesFormAction">
    <property name="PCManager" value="pcManager" />
    <property name="formObjectClass" value="...GeneratePC" />
    <property name="formObjectScope" value="FLOW" />
    <property name="formObjectName" value="pc" />
</bean>
```

Spring Web Flow – worth it?

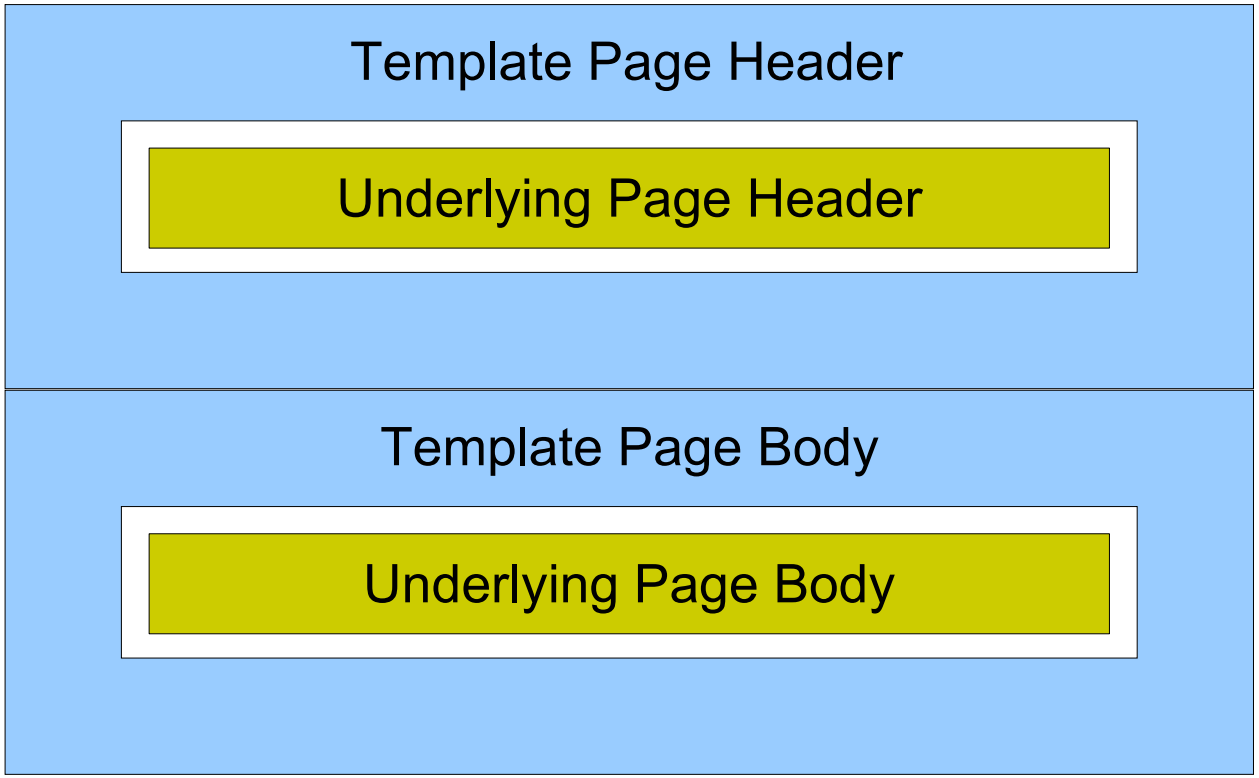
- Nice technology for handling flows
- Some disadvantages if you plan to use it for general-purpose pages
  - XML sprawl
  - Memory overhead if user navigates around within a single "flow" too long (e.g. if you model several independent pages with navigation as a "flow")
- Definitely easier than trying to code for all this behavior in a controller class
- I could go either way – but I favor it where the pages are be accurately represented as a flow

SiteMesh

- Curious that Spring MVC doesn't handle page templates/framing (because, uh, Rails does?)
- I need something to apply a standard header/footer/CSS look and feel, along with navigation tabs and so on
- Could just use JSP includes on every page, but that's a little intrusive and potentially hard to change
- SiteMesh acts as a filter in front of the web application, and applies templates based on URL patterns
- Works well to surround "the body" provided by the underlying web framework

SiteMesh Templates

- SiteMesh templates generate content that surrounds the underlying pages:



SiteMesh Configuration

- The SiteMesh configuration file says where the templates are stored, and which URL patterns go to which templates:

```
<decorators defaultdir="/WEB-INF/layouts">
    <!-- Touch screen template -->
    <decorator name="touchScreen" page="touchScreen.jsp">
        <pattern>/player/*</pattern>
    </decorator>
    <!-- Staff area template -->
    <decorator name="staffArea" page="staffArea.jsp">
        <pattern>/staff/*</pattern>
    </decorator>
</decorators>
```

SiteMesh Template JSP

- The template is a JSP with tags for inserting stuff from the underlying response:

```
<%@ taglib prefix="decorator"
    uri="http://www.opensymphony.com/sitemesh/decorator" %>
<html>
  <head>
    <title>Generate PC --
      <decorator:title default="Unnamed Page"/>
    </title>
    <decorator:head/>
  </head>
  <body>
    <p><i>This is some site header content</i></p>
    <decorator:body/>
    ...
```

SiteMesh – worth it?

- Great for sites where there's only one content area on the page, and the templating does not need to be configured at runtime
  - Tiles or Portlets, otherwise
- Not ideal architecture, perhaps – you know there's some extra buffering and parsing going on
  - Also a little difficult to set up navigation highlighting (must base on META tags declared in pages)
- But great results – templates and configuration are totally separated from underlying JSPs

Acegi Security

- Extremely modular and configurable security system, supporting web URL and component level security
- Handles form login, "remember me" cookies, hooking up to a security database/LDAP, etc. via configuration tweaks
  - Can also implement more advanced user data management and row-level authorization
- API for application to check current user, is in role, etc.
  - Also JSP tag library for the same
- Annotations for applying security to beans

Acegi Configuration

- Unfortunately, they have not adopted the Spring 2 configuration syntax, and the number and configuration of Acegi beans is, in a word, beastly
- I put it all in its own file to keep it from polluting the rest of the application configuration
- I changed things like the login URL, logout URL, login and error JSP names, default landing URL if you access the login page directly, post-logout URL, database queries to look up user password/groups, and URL to security role mapping
- Just about each of those is in a separate bean

Acegi Configuration Example

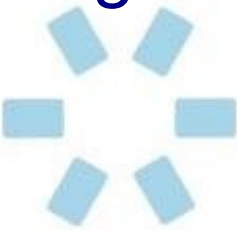
```
<bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">
  <property name="filterInvocationDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      PATTERN_TYPE_APACHE_ANT
      /**=httpSessionContextIntegrationFilter,logoutFilter,authenticationProcessingFilter,securityContextHolderAwareRequestFilter,anonymousProcessingFilter,
                                                                                    exceptionTranslationFilter,filterInvocationInterceptor
    </value>
  </property>
</bean>
<bean id="httpSessionContextIntegrationFilter" class="org.acegisecurity.context.HttpSessionContextIntegrationFilter"/>
<bean id="logoutFilter" class="org.acegisecurity.ui.logout.LogoutFilter">
  <constructor-arg value="/acegi-login.jsp"/>
  <constructor-arg>
    <list>
      <bean class="org.acegisecurity.ui.logout.SecurityContextLogoutHandler"/>
    </list>
  </constructor-arg>
  <property name="filterProcessesUrl" value="/acegi_logout_handler"/>
</bean>
<bean id="authenticationProcessingFilter" class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilter">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="authenticationFailureUrl" value="/acegi-login.jsp?login_error=1"/>
  <property name="defaultTargetUrl" value="/staff/playerView.do"/>
  <property name="filterProcessesUrl" value="/acegi_login_handler"/>
</bean>
<bean id="securityContextHolderAwareRequestFilter" class="org.acegisecurity.wrapper.SecurityContextHolderAwareRequestFilter"/>
<bean id="anonymousProcessingFilter" class="org.acegisecurity.providers.anonymous.AnonymousProcessingFilter">
  <property name="key" value="SomeSpecialValue"/>
  <property name="userAttribute" value="anonymousUser,Anonymous"/>
</bean>
<bean id="exceptionTranslationFilter" class="org.acegisecurity.ui.ExceptionTranslationFilter">
  <property name="authenticationEntryPoint">
    <bean class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilterEntryPoint">
      <property name="loginFormUrl" value="/acegi-login.jsp"/>
      <property name="forceHttps" value="false"/>
    </bean>
  </property>
  <property name="accessDeniedHandler">
    <bean class="org.acegisecurity.ui.AccessDeniedHandlerImpl">
      <property name="errorPage" value="/accessDenied.jsp"/>
    </bean>
  </property>
</bean>
<bean id="filterInvocationInterceptor" class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager">
    <bean class="org.acegisecurity.vote.AffirmativeBased">
      <property name="allowIfAllAbstainDecisions" value="false"/>
      <property name="decisionVoters">
        <list>
          <bean class="org.acegisecurity.vote.RoleVoter"/>
          <bean class="org.acegisecurity.vote.AuthenticatedVoter"/>
        </list>
      </property>
    </bean>
  </property>
  <property name="objectDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      PATTERN_TYPE_APACHE_ANT
      /staff/**=ROLE_Staff
      /**=IS_AUTHENTICATED_ANONYMOUSLY
    </value>
  </property>
</bean>
<bean id="authenticationManager" class="org.acegisecurity.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref local="dbAuthenticationProvider"/>
      <bean class="org.acegisecurity.providers.anonymous.AnonymousAuthenticationProvider">
        <property name="key" value="SomeSpecialValue"/>
      </bean>
    </list>
  </property>
</bean>
<bean id="dbAuthenticationProvider" class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="userDetailsService" ref="userDetailsService"/>
  <property name="userCache">
    <bean class="org.acegisecurity.providers.dao.cache.EhCacheBasedUserCache">
      <property name="cache">
        <bean class="org.springframework.cache.ehcache.EhCacheFactoryBean">
          <property name="cacheManager">
            <bean class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"/>
          </property>
          <property name="cacheName" value="userCache"/>
        </bean>
      </property>
    </bean>
  </property>
</bean>
```


Acegi Security – worth it?

- Much more configurable and somewhat more featureful than built-in Tomcat/J2EE security
- Doesn't require any app server configuration to work
- Works for URLs and Spring beans, but not EJBs
- Have to bend your mind around the XML configuration – most options you want are in there... somewhere
- User/group security model should only be used with care, but role-based security is the default
- I would use it again, but if standard J2EE security works fine for you, I won't argue

Briefly: Maven & HSQLDB

- Makes it easy to share/replicate the development environment
- Maven fetches all dependency JARs during the first build so they don't have to be checked in (and bumping the version is very easy)
- Maven creates IDE projects and runs tests and so on
- HSQLDB works great as a test database (in-memory or using a local directory)
- Overall, the code in CVS/SVN stays compact yet it's still easy to get the development environment running on a new machine



Conclusion

Closing Thoughts

- A lot of technologies
- Might do without Web Flow, Acegi, SiteMesh if app doesn't need the features they offer
- Still, a huge ton of features running in a "lightweight" servlet container
- Kind of makes you wonder what you need a full J2EE server for...
  - Avoid integrating so many things by hand?
  - Better management/configuration tools?

Benefit: Development Environment

- Very easy to get running on a new machine
- All you need is:
  - Tomcat or Jetty
  - Maven
  - A CVS or SVN client
  - (Plus I guess your favorite IDE)
- Great for demo or proof of concept or testing, using an embedded database

Benefit: Reduced Configuration

- There's still a lot of XML config, but...
  - It's virtually all in the Spring format (except flows, template mapping)
  - It's essentially reduced to a ton of one-time or boilerplate config, plus one entry per service or web URL/controller
  - Most of the meat is in annotations
- JPA is a huge step forward from EJB and older persistence engines
  - Still want a tool to code generate entities off the database

Greater than sliced bread?

- I find it faster to develop in this style than in most others
  - At least partly because IntelliJ supports it so well
- But I haven't used this on a performance-sensitive application
  - I do kind of wonder whether loading so many independent components into the "lightweight container" makes it heavier than a well-tuned app server
- I would use this architecture by default and consider alternatives if there's a good reason to

Could it be better?

- A lot of separate sources of documentation, of varying quality
- Still waste a fair amount of time waiting for the application to redeploy on code changes
  - *Did you go to the Rails sessions yesterday?*
- Could use some integrated tooling (JPA entity generator, Web Flow diagrammer, Acegi configuration editor, etc.)
- Best to heavily comment the configuration files, because you may not look at them often, but it's typically hard to remember the details much later

Q&A

Aaron Mulder
ammulder@chariotsolutions.com