



The State of Web Frameworks

Craig R. McClanahan
Senior Staff Engineer
Sun Microsystems, Inc.



Agenda

- **Background**
- **Variations on a theme**
- **Fundamental design patterns**
- **User interface components**
- **Frameworks and AJAX**
- **Summary**

Background

- **Web tier APIs were among the first standardization efforts outside the base Java Development Kit:**
 - **Servlet – Initially released in 1996**
 - **JavaServer Pages (JSP) – Initially released in 1999**
- **But the standards stopped at the foundations:**
 - **Low level abstraction of Hypertext Transfer Protocol**
 - **Easy mechanisms for combining static/dynamic markup**
- **They did not address application architecture issues**
- **Resulted in much open source software innovation**

Servlet API – The Foundation

- **Abstracting the basic concepts:**
 - **Servlet, Request, Response**
- **Adding a concept to deal with HTTP statelessness:**
 - **Session**
- **Later versions fleshed out basic functionality:**
 - **RequestDispatcher, Filter, Event Listeners**
- **It is *possible* to write applications with pure servlet APIs:**
 - `writer.println("<td>Customer Name:</td>");`
 - `writer.println("<td>" + cust.getName() + "</td>");`



Servlet API – The Issues

- **All of the code is in Java**
- **Markup generation is spread throughout the code**
- **Difficult to visualize the ultimate appearance**
- **Common look and feel hard to create**
- **Markup generation and business logic intermixed**

JSP 1.0 – Inside Out Servlets

- Even in dynamic applications, much content is static
- Servlets embed static *and* dynamic content in code
- What if we could embed dynamic content generation in static markup?
- JSP 1.0 supported three types of markers:
 - Variables (`<%! String foo; %>`)
 - Expressions (`<%= foo %>`)
 - Scriptlets (`<% foo = cust.getLastName() + " , " + cust.getFirstName(); %>`)



JSP 1.1/2.0 – Reduce Embedded Java

- **Embedded Java code still has issues:**
 - **Developers must be familiar with Java**
 - **Different syntax and semantics from JavaScript**
 - **Still intermixes markup and business logic**
- **JSP 1.1 – Custom Tags:**
 - **Page author deals with markup elements**
 - **Java code abstracted into separate classes**
- **JSP 2.0 – Addresses even more issues**
- **But JSP had a hard reputation to shake (because of scriptlets)**

Web Application Frameworks

- **While standards were evolving, innovative solutions were explored:**
 - **Application architecture frameworks**
 - **User interface component models**
- **To meet specific needs:**
 - **“Hello, World” does not help build real world apps**
 - **Many newcomers to webapps were also new to Java**
- **By early 2000s, roughly 50 available choices**
- **How does an architect decide what to use?**



Variations On A Theme

- **When you step away from the details:**
 - **Most frameworks deal with the same set of issues**
 - **Much overlap in how these issues are addressed**
- **Selecting a framework means:**
 - **Accepting the combination of architectural decisions made by the designers of the framework**
- **What issues are important in web applications?**



Variations On A Theme

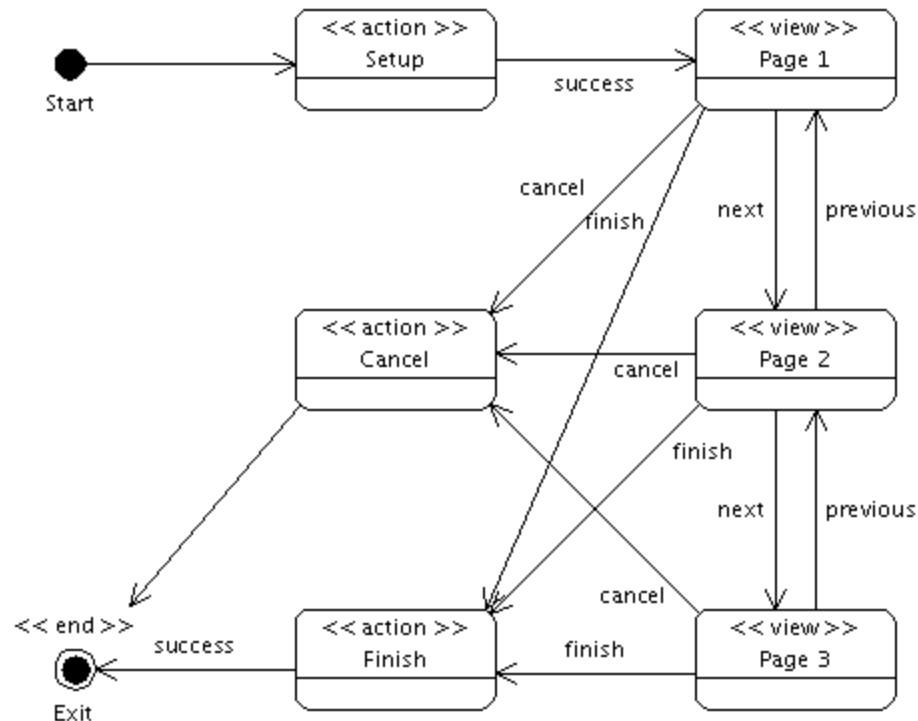
- **Key architectural decisions for web frameworks:**
 - **Modelling of page navigation decisions**
 - **Provisions for accessing model tier data**
 - **Representation of static and dynamic markup**
 - **Mapping incoming requests to business logic**
 - **Existence (or not) of a user interface component model**
- **We will briefly review the first three decisions**
- **The latter two are more interesting and deserve a deeper look**

Sidestep: Terminology

- Most web application frameworks define their behavior in terms of the *model-view-controller* (MVC) design pattern
- For purposes of our discussion:
 - Model – Persistent data and business logic
 - In large applications, often subdivided into separate tiers
 - View – The interface with which the user interacts
 - HTML and JavaScript (humans) or services (programs)
 - Controller – Management software that does *mappings*
 - Incoming requests to business logic
 - Page navigation decisions to corresponding view artifacts

Modelling Page Navigation Decisions

- Many architects start the design process by drawing a “storyboard” -- or a UML *State Diagram*



Created with Poseidon for UML Community Edition. Not for Commercial Use.



Modelling Page Navigation Decisions

- **Destination based navigation:**
 - Source view knows the name or URL of the destination
 - Example: Tapestry action listener can
 - Navigate to a specific URL
 - Be injected with a page object representing the destination
 - Example: Spring MVC *ModelAndView* return value
- **Outcome based navigation:**
 - Source view returns a *logical outcome*
 - External configuration information defines rules
 - Example: Struts returns *ActionForward*
 - Example: JSF action methods return outcome string

Accessing Model Tier Resources

- **Most frameworks are agnostic here**
- **This is the mark of good architectural design:**
 - **Web application architecture should not dictate architecture of business logic or persistence strategy**
 - **Allow existing implementations to be reused**
 - **Encourages tier-specific unit tests**
- **Typical options include:**
 - **Java2 Enterprise Edition (J2EE) – EJB, DataSource, ...**
 - **Dependency injection framework – Spring, HiveMind, ...**
 - **Specialized persistence tiers – Hibernate, TopLink, ...**

Representing Static/Dynamic Markup

- **Most frameworks support JSP for this:**
 - **Static markup is simply entered inline**
 - **Dynamic markup entered with custom tags, expressions**
- **Tapestry is an interesting exception ...**
- **Many frameworks also support integration with non-JSP technologies:**
 - **Templating systems (Velocity, Freemarker, ...)**
 - **XML documents transformed by XSLT (Cocoon)**

Representing Static/Dynamic Markup

- Example: Login page using JavaServer Faces:

```
<h:form>
  <table border="0">
    <tr><td>Username:</td>
      <td><h:inputText      id="username"
                            value="#{logon.username}" /></td></tr>
    <tr><td>Password:</td>
      <td><h:inputSecret   id="password"
                            value="#{logon.password}" /></td></tr>
    <tr><td><h:commandButton id="logon"
                            action="#{logon.authenticate}" /></td></tr>
  </table>
</h:form>
```

Representing Static/Dynamic Markup

- **Tapestry takes a different approach:**
 - **Entire page represented in (almost) pure HTML**
 - **Dynamic content identified by HTML elements with a *jwcid* attribute representing component id and type**
 - **When rendered, dynamic content replaces this element**
- **Enables two modes during development:**
 - **Static view – display the template**
 - **Dynamic view – execute the template**
- **JavaServer Faces can do the same with plugins:**
 - **Facelets – java.net project**
 - **Clay – Shale framework component**

Representing Static/Dynamic Markup

- Example: Login page using Tapestry:

```
<form jwcid="form@Form" success="listener.doLogon">
  <table border="0">
    <tr><td>Username:</td>
      <td><input      jwcid="username@TextField"
                    value="ognl:username" /></td></tr>
    <tr><td>Password:</td>
      <td><input      jwcid="password@TextField"
                    hidden="true"
                    value="ognl:password" /></td></tr>
    <tr><td><input type="submit" value="Logon"></td></tr>
  </table>
</form>
```



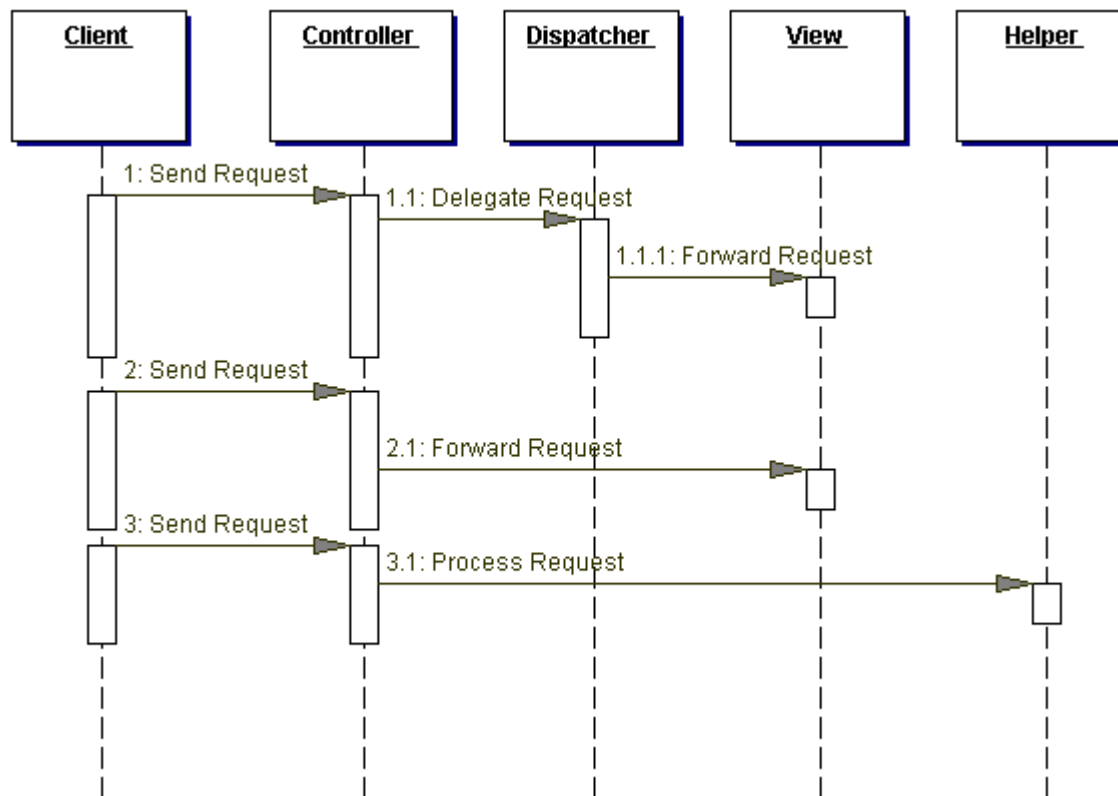
Sidestep: Design Patterns

- A common mechanism for understanding architectures is to examine the *design patterns* that are implemented
- A seminal book popularized the term:
 - Gamma, Helm, Johnson, Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1977
- In the J2EE space, a companion volume is valuable:
 - Alur, Crupi, Malks, Core J2EE Patterns: Best Practices and Design Strategies, Prentice Hall, 2001
- Two patterns are very popular foundations for web application frameworks



Front Controller Design Pattern

- “Use a controller as the initial point of contact ...”





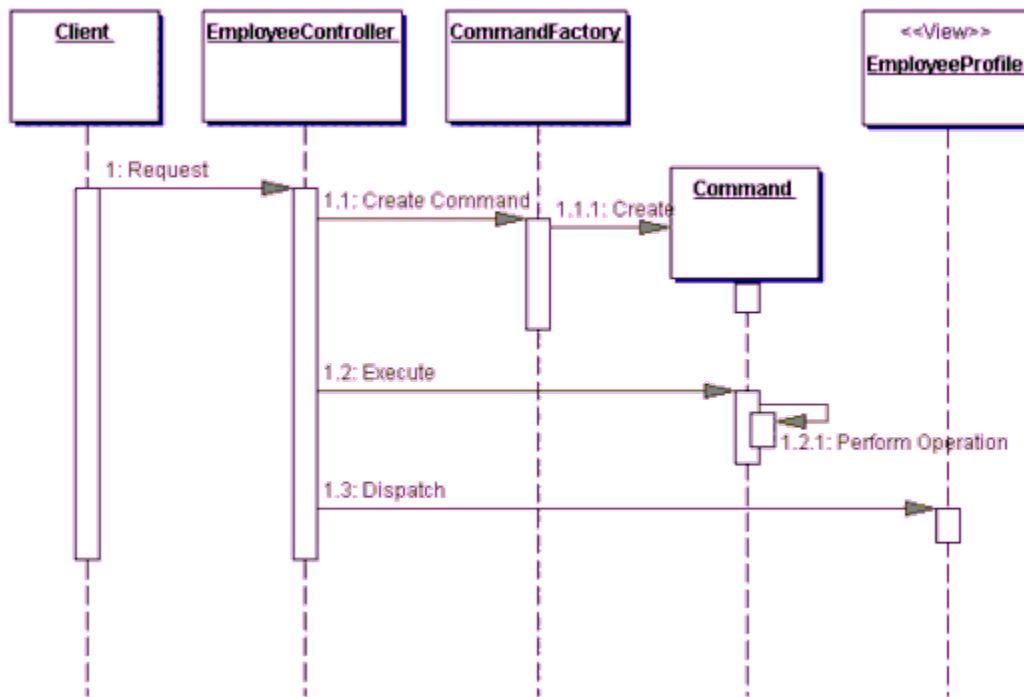
Front Controller Design Pattern

- **Participants and responsibilities:**
 - ***Controller*** – Initial contact point (may delegate to helpers)
 - ***Dispatcher*** – Responsible for view management, navigation
 - ***View*** – Represents and displays information to client
 - ***Helper*** – Assistant to controller or view
- **Consequences of this design pattern include:**
 - ***Centralizes control*** – Central place to customize
 - ***Improves security management*** – Centralized administration
 - ***Improves partitioning, reuse, and maintainability*** – Encourages clean separation of business and view logic



Front Controller Design Pattern

- Patterns can be implemented by a variety of strategies, such as **Command and Controller**:

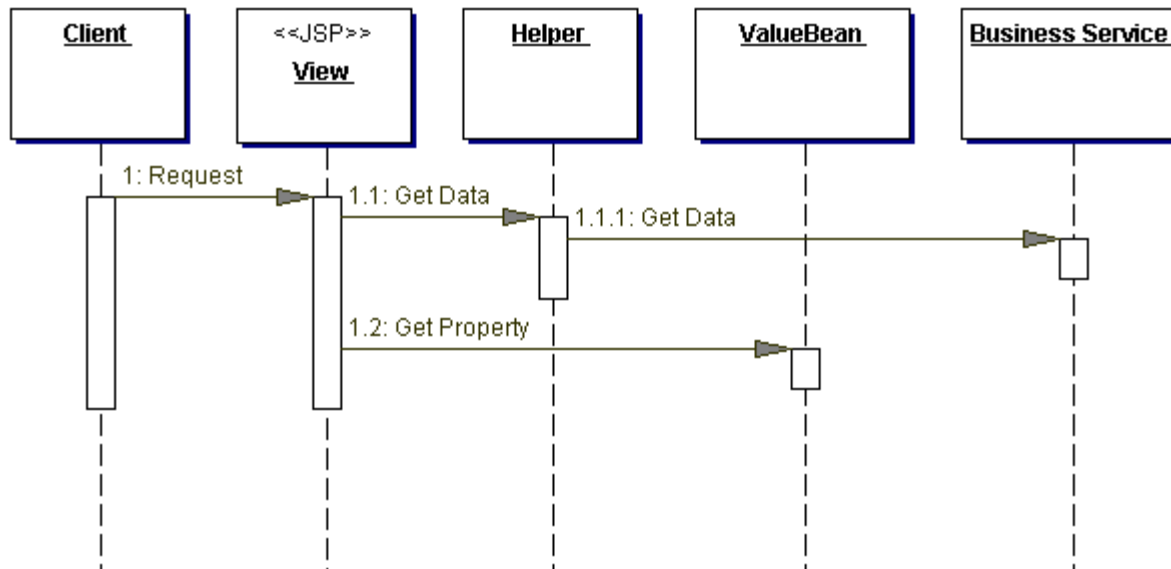


- Some frameworks use this approach to *decorate* business logic



View Helper Design Pattern

- “Use a view as the initial point of contact ...”





View Helper Design Pattern

- **Participants and responsibilities:**
 - ***View*** – Represents and displays information to client
 - ***Helper*** – Assistant to controller or view
 - ***Value Bean*** – Specialized helper responsible for intermediate model state
 - ***Business Service*** – Business logic to be accessed
- **Consequences of this design pattern include:**
 - ***Improves partitioning, reuse, and maintainability*** – Encourages clean separation of business and view logic
 - ***Improves role separation*** – Reduces dependencies between tiers, improves unit testability
- **Also, more familiar to some Java newcomers**



But I Want *Both* Sets of Benefits

- The two patterns share a common consequence:
 - *Improves partitioning, reuse, and maintainability*
- But they also feature unique advantages:
 - Front controller pattern:
 - *Centralizes control*
 - *Improves manageability of security*
 - View helper pattern:
 - *Improves role separation*
- Ideal framework would allow a combination of benefits
 - But how can we do that? The development models look totally different?



One Application – Two Approaches

- **Struts has always included a canonical example:**
 - **The “Mail Reader” application**
- **Let's examine an implementation in two styles:**
 - ***Front controller* – Using Struts 1.2**
 - ***View helper* – Using JSF 1.1 and Shale**
- **First, we will look at the runtime behavior ...**
- **Next, we will compare some source artifacts:**
 - **Both versions share a common persistence tier**
 - **Functionality of both applications is identical**



One Application – Two Approaches

- **Struts solution complexity metrics:**
 - 6 JSP pages, 10 Java classes
 - Complex configuration metadata (form beans, wildcard URL mappings)
 - *Action* classes separate from form beans
 - A WebWork version would look much like the JSF approach
- **JSF + Shale complexity metrics:**
 - 6 JSP pages, 8 Java classes (no form beans)
 - Straightforward configuration metadata (managed beans, navigation rules)
 - *Action* classes have properties for intermediate view state



One Application – Two Approaches

- **Some common characteristics emerge as well**
- **Complexity of JSP pages is roughly the same:**
 - **Struts HTML tags, JSF UI component tags**
- **Complexity of individual action methods is roughly the same:**
 - **Pull data from request, call business logic**
- **Overall “shape” of the application is roughly the same:**
 - **Fundamental design patterns are an internal implementation detail of the framework**
 - **Does not represent, by itself, a reason to choose one type of framework over another**



Extending The Application

- **What about the impact of adding a new feature?**
 - **Ensure user is logged on before page can be accessed**
- **In a *front controller* framework like Struts:**
 - **Individual check in each JSP page (not recommended)**
 - **Container managed security or servlet filter**
 - **Customize Struts RequestProcessor implementation**
- **In a *view helper* framework like JSF:**
 - **Individual check in each JSP page (not recommended)**
 - **Container managed security or servlet filter**
 - **Customize default JSF ActionListener before calling action**
 - **Add a JSF PhaseListener to perform the check**

Extending The Application

- **What about the impact of adding a new feature?**
 - **Enforce a common look and feel with banner, navigation bar**
- **In a *front controller* framework like Struts:**
 - **Hard code each JSP page to match (not recommended)**
 - **Post-processing filter like SiteMesh**
 - **Integrated layout management like Tiles**
- **In a *view helper* framework like JSF + Shale:**
 - **Hard code each JSP page to match (not recommended)**
 - **Post-processing filter like SiteMesh**
 - **Integrated layout management like Tiles**



Extending The Application – Lessons

- The underlying architecture of the framework:
 - Is interesting (perhaps more so to framework geeks :-)
 - Influences what kinds of customizations are easy
 - Does not *a priori* dictate what kinds of customizations are possible
 - Might not be a good primary reason to pick a framework
- Why else might I choose one framework over another?

Framework Differentiation

- We have examined one fundamental distinguishing characteristic between frameworks:
 - The fundamental design pattern that is implemented
- The second fundamental distinguishing characteristic is whether a *user interface component model* is used:
 - In a Swing app, there is no question ... components are it
 - Would *you* want to use `java.awt.Canvas` (bitmap) directly?
- Web applications evolved from a world where it was initially required that you could handwrite markup:
 - Web browsers immature, not standardized
 - Limited or no development tools



Framework Differentiation

- Culture evolved that *page designer* was totally in charge of every single detail of visual appearance:
 - Customized HTML development tools emerged
 - Tools did “code generation” of markup to match the desires of the designer
- But what works for web sites does not always work for web applications:
 - Consistent look and feel is very important
 - So is reusability and maintainability
 - Do not always have the luxury of a graphic artist to work on the look and feel



User Interface Components

- **Some frameworks define and use UI components**
- **UI components take responsibility for:**
 - **Rendering the appropriate visual representation**
 - **Maintain state of user's interaction with the component**
 - **Perform validations (correctness checks)**
 - **Convert input (typically Strings) to correct model data types**
 - **Typically *bound* to model tier data**
- **Basically the same responsibilities in web application and rich client spheres**



User Interface Components Models

- A user interface component model also supports:
 - Hierarchical relationships between components
 - *Layout* components that take responsibility for children
 - Ability to dynamically modify the entire component tree
- Example: DHTML lets you manipulate client DOM dynamically
- Example: JSF based “SQL Browser” application
 - Allows user to type an SQL *SELECT* statement
 - Dynamically create visible columns based on which database columns were returned
- Demo of SQL Browser in action ...



User Interface Components Models

- ***Self describing components*** – Useful for tools
- **Examples of component description:**
 - **Localized display name (for the palette)**
 - **Tooltip text to show when mouse hovers**
 - **Popup help displayed on demand**
 - **List of properties to be included in the property sheet**
 - **Customized property editors**
 - **Design time behavior when user interacts with components**
- **Result** – you can provide a graphical Interactive Development Environment tool like Java Studio Creator ...
 - <http://developers.sun.com/jscreator/>

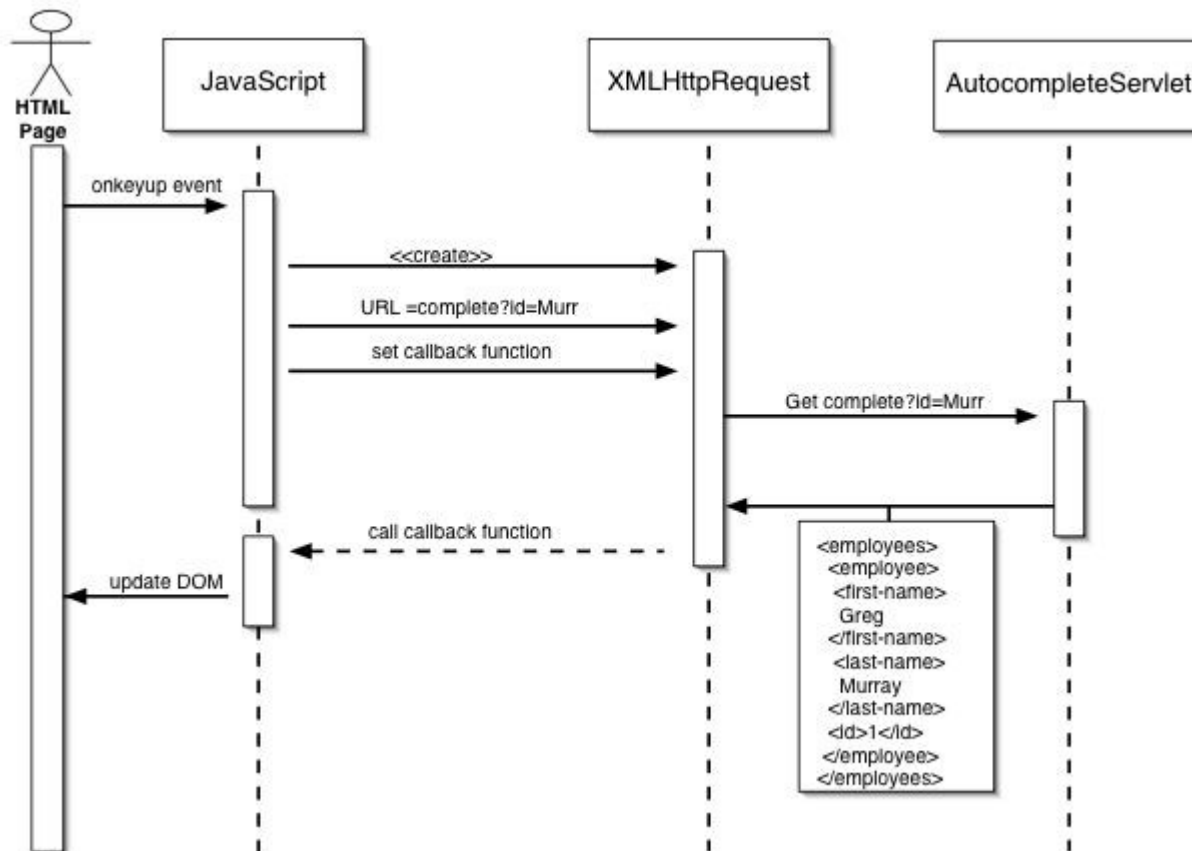
Asynchronous XML and JavaScript

- Impossible to ignore the hype around AJAX recently
- Techniques are not actually new:
 - XMLHttpRequest in Internet Explorer for years
 - Alternative techniques (hidden frames) even longer
- What is new is a synergy:
 - Reasonably portable XMLHttpRequest implementations
 - Robust DHTML and JavaScript implementations
- Coupled with an increased desire for interactive web UIs
- What does this mean for framework architectures?
 - One option is to just ignore it



AJAX Based Auto Complete Text Field

- No framework involved beyond Servlet API
 - But requires developer to deal with JS/DHTML directly



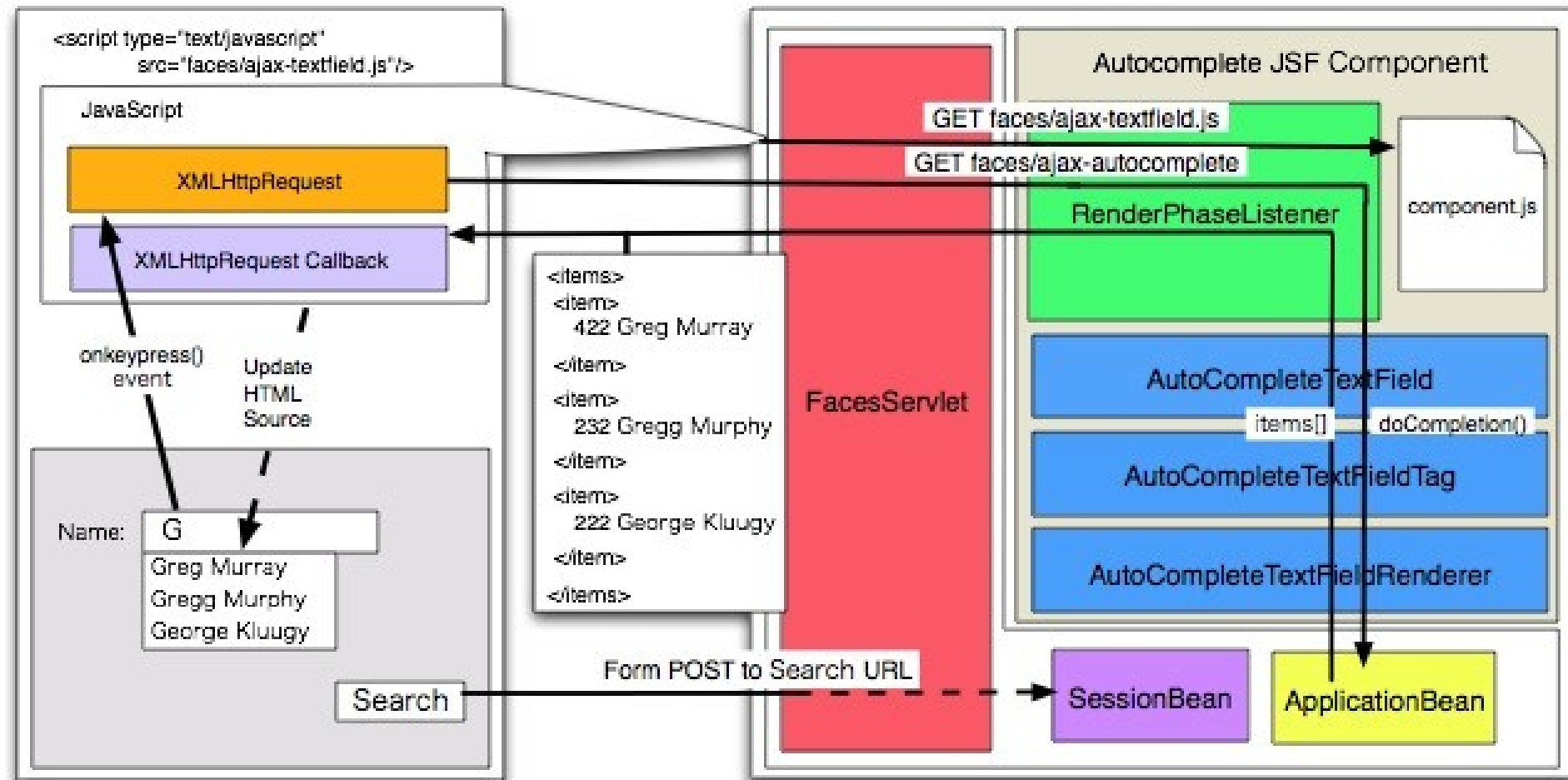
Framework Value Added Features

- **Client-server integration services:**
 - **WebWork** – Client side validation, partial page refresh
 - **Shale** – Serve static resources, map dynamic requests
- **Encapsulate AJAX behavior in “widgets”:**
 - **WebWork** – Tabbed panel with asynch content loading
 - **Tapestry** – Modal dialog, tree control
 - **Apache MyFaces** – AJAX enabled JSF components
 - **BluePrints Catalog** – AJAX enabled JSF components
- **How would you implement an Auto Complete Text Field as a JSF component?**

Auto Complete Text Field Component

Client

Java EE





Auto Complete Text Field Component

- **Because we created a component, we can leverage development tools**
- **Demo of Auto Complete Text Field component ...**
- **Benefits of wrapping AJAX functionality in components:**
 - **Isolates application developer from complex JavaScript and Dynamic HTML interactions**
 - **Allows developer to focus on model tier interactions**
 - **Provides familiar component oriented development paradigm (same as non-AJAX components)**
 - **Enables tools to support development of AJAX based applications**



Summary

- **Web application frameworks became popular because:**
 - **Addressed usability limitations in low level standard APIs**
 - **Encouraged better architectures by separating concerns**
 - **Provided structure to focus on individual elements**
- **Web application frameworks address common issues:**
 - **Modelling of page navigation decisions**
 - **Provisions for accessing model tier resources**
 - **Representation of static and dynamic markup**
 - **Mapping incoming requests to business logic**

Summary

- **Key distinguishing characteristics:**
 - **Fundamental design pattern used internally:**
 - Typically *Front Controller* or *View Helper*
 - May or may not impact what you can do with the framework
 - **Support (or not) for a user interface component model**
- **It is possible for a framework to implement characteristics of both design patterns:**
 - **And therefore provide both sets of benefits**
- **JavaServer Faces is one such framework:**
 - **Also includes extension points for adding more services**
 - **Examples: Shale, Seam**



Questions