

The State of Web Frameworks

Craig R. McClanahan
Sun Microsystems, Inc.
Craig.McClanahan@sun.com
February 25, 2006

Explore the current state of web application frameworks for Java^(tm) technology based applications

- 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 of the base Java Development Kit (JDK)
 - 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 (HTTP)
 - Easy mechanisms for combining dynamic and static markup
- And they did not address application architecture or user interface components
 - At least until JavaServer Faces – initially released in 2004
- This resulted in **much innovation in the Open Source Software space**

- Abstracting the basic concepts:
 - Servlet, HttpServletRequest, HttpServletResponse
- Adding a concept to deal with HTTP statelessness:
 - HttpSession
- Later versions fleshed out the basic functionality
 - RequestDispatcher – Composing requests from multiple pieces
 - Filter – Command pattern decoration of processing components
 - Event Listeners – Simple lifecycle management
- It is possible to write apps with just servlets:
 - `writer.println("<td>Customer Name:</td>");`
 - `writer.println("<td>" + customer.getName() + "</td>");`
- But this approach has several issues

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

- Even in a dynamic web application, much of the content is actually static
- Servlets embed static **and dynamic** content generation in code:
 - `writer.println("...");`
- 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 = customer.getLastName() + " " + customer.getFirstName(); %>`)

- But embedded Java code still has issues:
 - Still requires familiarity with the Java language
 - Syntax and semantics very different from client side scripting languages (JavaScript) also present in the pages
 - Still intermixes markup and business logic
- JSP 1.1 provides custom tags:
 - Page author deals with markup elements
 - Java code abstracted into separate classes (tag handlers)
 - JSP Standard Tag Library (JSTL) for common use cases
- JSP 2.0 – released in 2003 – addresses even more of these issues
- But JSP's reputation for problems due to intermix could not be easily shaken

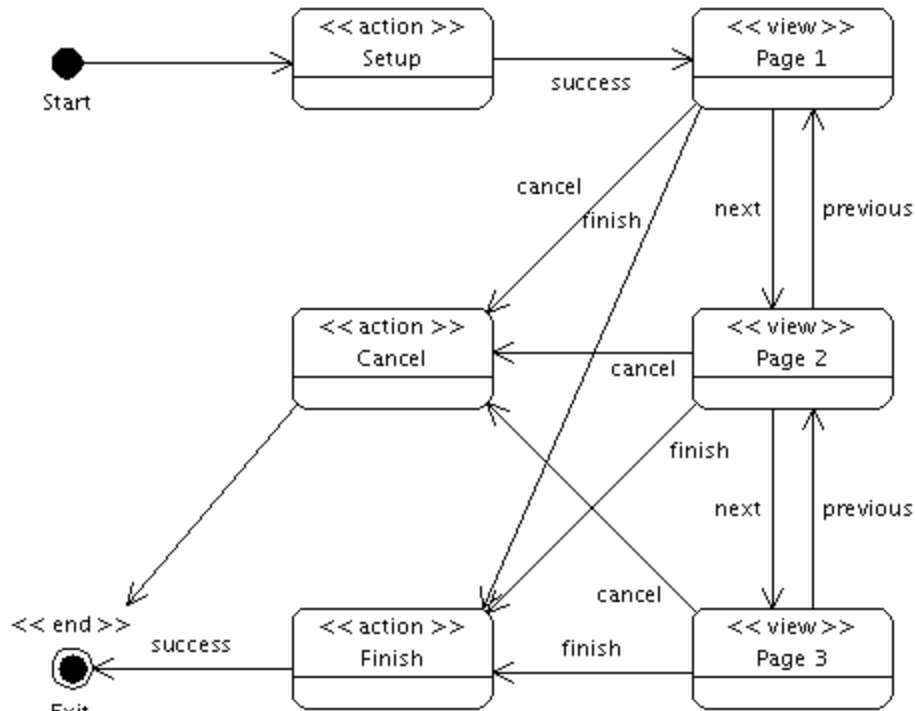
- While the standards were evolving, innovative solutions were explored
 - Application architecture frameworks
 - User interface component models
- To meet specific needs:
 - “Hello, World” examples do not help developers build real applications
 - Most developers did not wish to deal with low level server functionality
 - Many people building web applications were newcomers to Java, as well as newcomers to the web
- By the early 2000s, there were roughly 50 available examples in the open source arena:
 - A few widely used, many used by small groups, and lots of one-person projects
 - Commercial software packages provided frameworks in addition, and/or supported de facto standard open source frameworks
- How does an application architect decide what to use?

Variations On A Theme

- When you step away from the nitty gritty details:
 - Most frameworks deal with the same sets of issues
 - There is much overlap in the way these issues are addressed
- Selecting a framework means accepting the combination of architectural decisions made by the designers of that framework
- Key architectural decisions defined by web application frameworks:
 - Modelling of page navigation decisions
 - Provisions for accessing model tier resources
 - 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 last two decisions are more interesting, and deserve more attention

- Many web application frameworks define their behavior in terms of the model-view-controller (MVC) design pattern
 - Popular in Rich Client applications since the 1980s
 - Web applications implement the pattern slightly differently, due to HTTP latency
- For the purposes of our discussion:
 - Model – Persistent data and the business logic that processes it
 - In large applications, often subdivided into separate tiers
 - View – The interface with which the user directly interacts
 - For web applications, typically embodied in HTML and JavaScript
 - Controller – Management software that performs *mappings*
 - *Incoming requests to corresponding business logic (model tier) components*
 - *Page navigation decisions to corresponding view tier components*

- Many application architects start the design process by drawing a “storyboard” or “page flow” diagram
- Unified Modelling Language (UML) has a *State Diagram* type that is commonly used (formally or informally) to represent the logical flow



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

- **Destination based navigation:**
 - Implementation of source view knows the name or identity of the destination view for each transition, and says “go to this page next”
 - Example: In Tapestry, action listeners can return:
 - Void – Stay on same page
 - String – Navigate to the specified URL
 - *IPage* – Instance of the object representing the next page to be executed
 - *Spring MVC follows the third pattern, with a ModelAndView return value*
- **Outcome based navigation:**
 - *Source view returns an “outcome” value that says “this is what happened”*
 - *External configuration information defines actual navigation that occurs*
 - *Example: In Struts, an Action returns a logical ActionForward object*
 - *Name of the ActionForward is used to look up name of the next view*
 - *Logical outcomes can be defined locally (to a particular Action) or globally*
 - *JavaServer Faces and WebWork follow a similar pattern, with a String return value*

- Most frameworks are agnostic about model tier technologies they support
- This is the mark of good architectural design:
 - Architecture of the web application should not dictate architecture of the business logic and persistence tiers
 - Allows existing implementations to be leveraged
 - Encourages tier-specific unit tests in addition to system integration tests
- Typical options include:
 - Java2 Enterprise Edition (J2EE) standard resources (EJB, Data Sources, Web Services)
 - Dependency Injection Frameworks (Spring, HiveMind, PicoContainer)
 - Specialized persistence tier implementations (Hibernate, Toplink, etc.)
- Web architectures based on JavaServer Faces may leverage the dependency injection functionality of the Managed Beans feature

- Most frameworks support the use of JavaServer Pages (JSP) to represent static and dynamic markup
 - Static markup is simply entered inline
 - Dynamic markup is entered by a combination of:
 - JSP custom tag handlers that emit markup
 - In JSP 2.0, custom tag handlers can be created with JSP tag files as well as with Java code
 - Expression Language (EL) expressions in tag attributes or inline text
 - Use of Java runtime expressions and scriptlets is generally frowned upon
- Tapestry is an interesting exception, as we will see later
- Many frameworks also support integration with non-JSP technologies
 - Velocity and Freemarker templating systems (Struts, WebWork, Spring MVC, ...)
 - XML documents transformed by XSLT stylesheets (Cocoon, ...)

- Example: In JavaServer Faces, a login page might look like this:

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

- Tapestry takes a different approach:
 - Entire page is represented in (almost) pure HTML
 - Dynamic content is identified by HTML elements with a **juwid attribute**
 - **This attribute represents a component identifier and type**
 - **When rendered, the dynamic output from the component replaces the original element**
- **Enables two modes during development:**
 - **Static view – display the template**
 - **Developer and end user review the visual appearance**
 - **Dynamic view – execute the template**
 - **Developer and end user validate the dynamic behavior**
- **With available extensions, the same approach is possible with JavaServer Faces technology:**
 - **Struts Shale project -- “Clay” plugin**
 - **Facelets project**

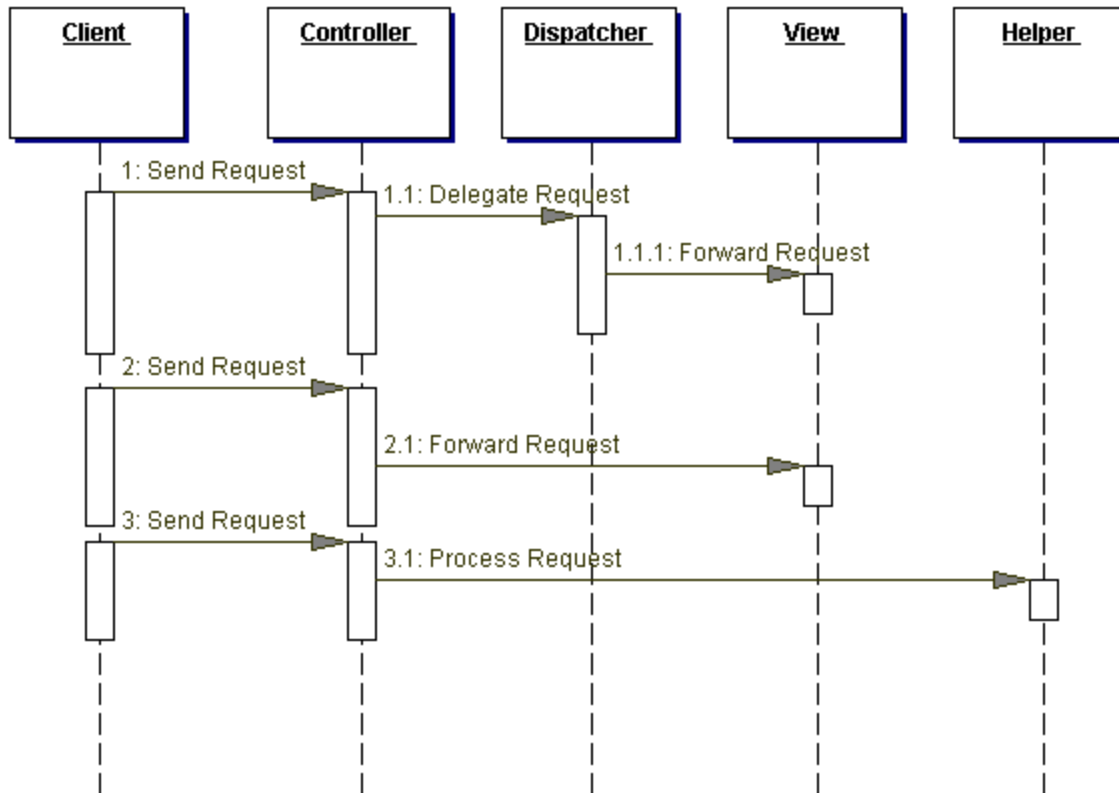
- Example: In Tapestry, a login page might look like this:

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

Fundamental Design Patterns

- A common mechanism for understanding architectures is to examine the *design patterns that are implemented*
- A seminal book popularized this term (the “Gang of 4” book):
 - Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1977.
- In the Java2 Enterprise Edition space, a companion volume is equally valuable:
 - Alur, Crupi, Malks, *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall, 2001
 - <http://java.sun.com/blueprints/corej2eepatterns/index.html>
- Two such patterns are very popular among web application frameworks
 - Front Controller (Struts, WebWork, Spring MVC, ...)
 - View Helper (JavaServer Faces, Tapestry, Microsoft ASP.Net, ...)
- It is also possible to implement a combination of these patterns

- “Use a controller as the initial point of contact for handling a request. The controller manages the handling of the request, including invoking security services such as authentication and authorization, delegating business processing, managing the choice of the appropriate view, handling errors, and managing the selection of content creation strategies”

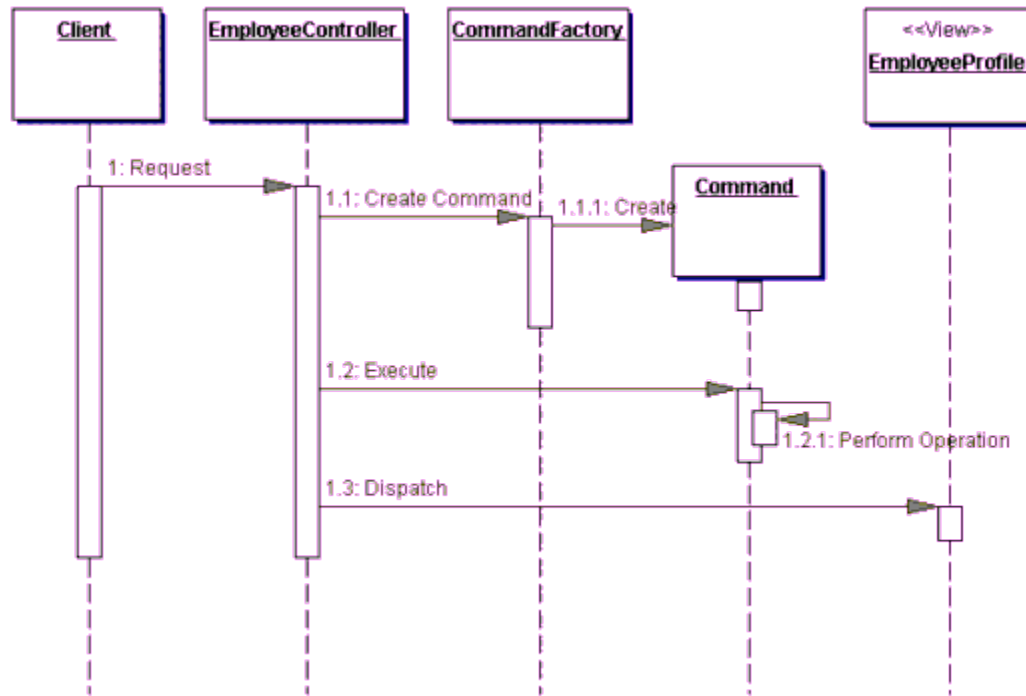


- The *Front Controller pattern has these participants and responsibilities:*
 - *Controller – Initial contact point for all requests (may delegate to helpers)*
 - *Dispatcher – Responsible for view management and navigation*
 - *View – Represents and displays information to the client*
 - *Helper – Assistant for a controller or a view in fulfilling their responsibilities*

- *Consequences of this design pattern include:*
 - *Centralizes control – A central place to handle system services, such as resource allocation and cleanup.*
 - *Improves manageability of security – Central location for access and authentication.*
 - *Improves partitioning, reuse, and maintainability – Encourages cleaner separation of presentation logic and business logic. Reduces impact of changes in one tier from affecting logic in other tiers.*

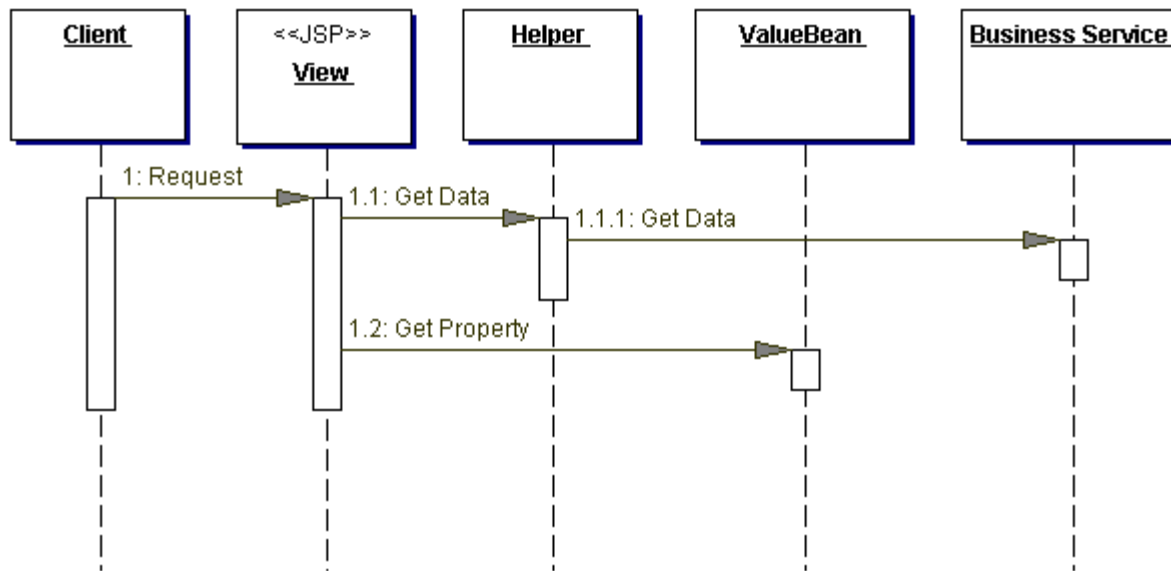
The Front Controller Design Pattern

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



- Some frameworks use this approach to decorate the application business logic that is ultimately invoked with extra services and functionality

- “A view contains formatting code, delegating its processing responsibilities to its helper classes, implemented as JavaBeans or custom tags. Helpers also store the view's intermediate data model and serve as business data adapters.”



- The *View Helper* pattern has these participants and responsibilities:
 - *View* – Represents and displays information to the client
 - *Helper* – Assistant for a controller or a view in fulfilling their responsibilities
 - *ValueBean* – Specialized helper that is responsible for intermediate model state
 - *BusinessService* – The business logic that the client is seeking to access

- Consequences of this design pattern include:
 - Improves partitioning, reuse, and maintainability – Encourages cleaner separation of presentation logic and business logic. Reduces impact of changes in one tier from affecting logic in other tiers.
 - Improves role separation – Separating logic reduces dependencies between tiers, and improves unit testability of individual components.

- The two patterns we have examined 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*
- An ideal framework would allow the application to benefit from the combined set of consequences
- But the development models look totally different! How can we do that?
 - Let's take a deeper look ...

- Struts has always included a canonical example application
 - The “Mail Reader” application
- We will examine this application implemented in both styles:
 - *Front Controller design pattern, using Struts 1.2*
 - *View Helper design pattern, using JavaServer Faces 1.1 and Shale 1.0.0*
- *First, let us examine the runtime behavior of this application*
 - *As implemented with Struts ...*
 - *As implemented with JavaServer Faces and Shale ...*
- *Next, let us compare some of the source code artifacts ...*
 - *Both versions share a common model tier implementation*
 - *org.apache.struts.apps.mailreader.dao*
 - *Functionality of both web applications is identical*

- View Tier – JSP Page
 - Struts version ...
 - JavaServer Faces + Shale version ...
- View Tier – Backing Bean
 - Struts version (form bean) ...
 - Struts version (action) ...
 - JavaServer Faces + Shale version ...
- Controller Tier – Configuration Information
 - Struts version ...
 - JavaServer Faces + Shale version ...

- Struts Solution – Complexity Metrics:
 - 6 JSP pages, 10 Java classes
 - Complex configuration metadata (form beans, wildcard URL mapping)
 - *Action classes separate from form beans*
 - *A WebWork version would look much more like the JavaServer Faces version*
- JavaServer Faces + Shale Solution – Complexity Metrics:
 - 6 JSP pages, 8 Java classes
 - Straightforward configuration metadata (managed beans, navigation rules)
 - *Action classes have properties for the intermediate view state*
- Complexity of JSP pages roughly the same
- Complexity of individual action methods roughly the same
- Overall application organization roughly the same

- Impact of adding a feature to this application
 - Ensure user is logged on before a page can be accessed
- In a *front controller framework like Struts, you could do this:*
 - *Individual check on each JSP page (not recommended)*
 - *Container managed security*
 - *Servlet filter*
 - *Customize the Struts RequestProcessor implementation*
 - *In Struts 1.3 or WebWork, this would be adding a “command” or “interceptor”*
- In a *view helper framework like JavaServer Faces you could do this:*
 - *Individual check on each JSP page (not recommended)*
 - *Container managed security*
 - *Servlet filter*
 - *Add a JavaServer Faces PhaseListener to perform the check*

- Impact of adding a feature to this application
 - Enforce a common look and feel with banner and navigation bar
- In a *front controller framework like Struts*, you could do this:
 - *Hand code each page to match the style guidelines (not recommended)*
 - *Post-processing filter like SiteMesh*
 - *Integrated layout management system like Tiles*
- In a *view helper framework like JavaServer Faces* you could do this:
 - *Hand code each page to match the style guidelines (not recommended)*
 - *Post-processing filter like SiteMesh*
 - *Integrated layout management system like Tiles*
 - Acquire or create layout management components
- Summary: while the underlying architecture of the framework might be interesting, it does not necessarily dictate what you can do with it
- So, why else might I choose one framework over another?

User Interface Components

- **We have examined one of the two fundamental distinguishing characteristics of web application framework architectures**
 - Fundamental design pattern that is implemented
- **The second fundamental distinguishing characteristic is whether the framework implements a user interface component model or not**
 - In a Swing application there is no question ... it is components or nothing
 - **Would you want to program to the `java.awt.Canvas` or `java.awt.Graphics` APIs?**
- **In web applications, HTML evolved in an environment where it was required that developers could hand-author markup**
 - **Web browsers were immature and not yet standardized**
 - **Initially, there were no development tools**
 - **Evolution of development tools was around the idea that the page author was totally in charge of every single detail of the visual appearance**
- **What works for web sites does not necessarily work for web apps**
 - **Consistent look-and-feel is very important**
 - **So is reusability and maintainability**

- In addition to the differences in design patterns implemented, a *fundamental difference in web application frameworks is whether or not they implement the notion of a user interface component model*
- *What is a user interface component?*
 - *Same in the web application sphere as in the rich client sphere*
 - *A user interface component takes responsibility for:*
 - *Rendering the appropriate markup to visualize this component*
 - *Maintain state of the user's interaction with the component*
 - *Performing validation (correctness) checks on user input values*
 - *In the case of a validation error, users expect that the application will rerender the incorrect input values so that they can be repaired*
 - *Converting the string input from a web client to the appropriate model data type*
 - *Components are typically bound to model tier data values*

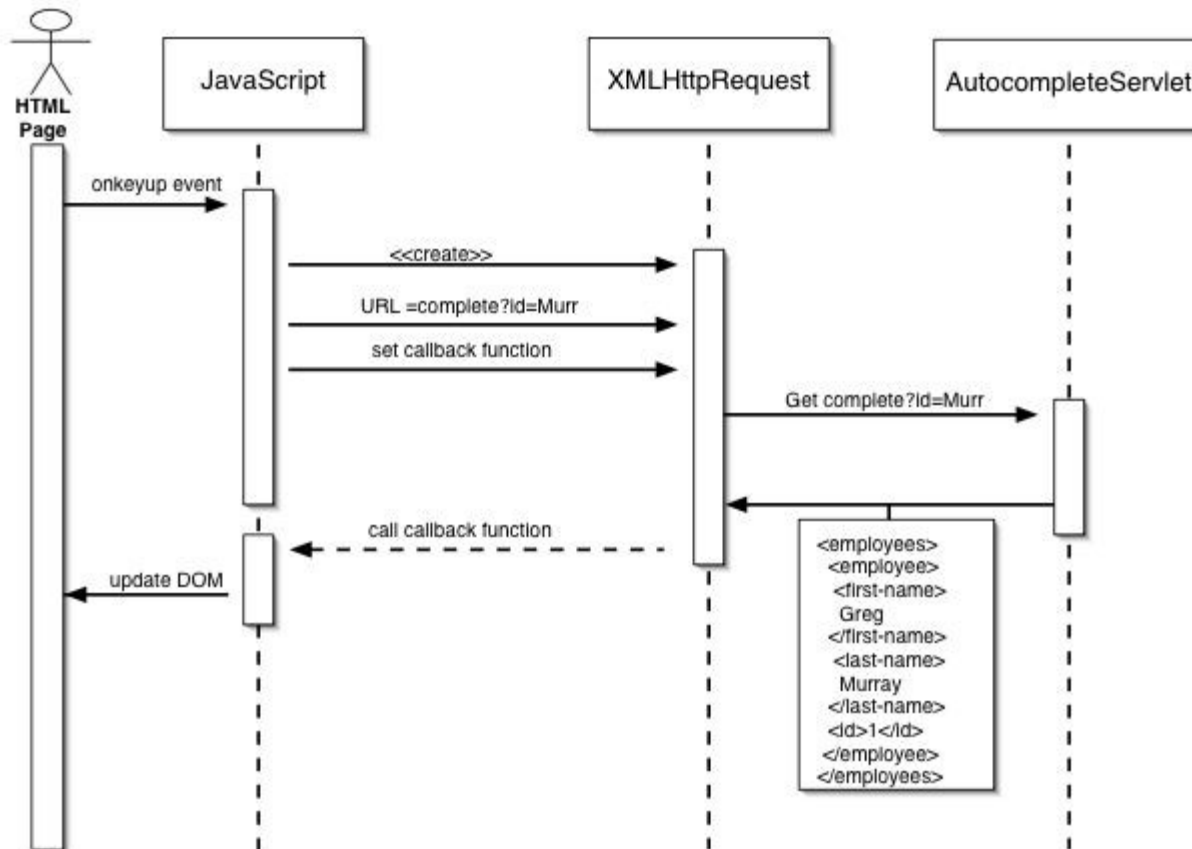
- *A user interface component model also supports:*
 - *Hierarchical relationships between user interface components*
 - *Typically represented in a “component tree”*
 - *“Layout” components that take responsibility for the overall visual arrangement of the individual user interface components*
 - *Ability to dynamically modify the set of components that comprise a view*
- *Example – A SQL Browser application with JavaServer Faces*
 - *Allow the user to specify an SQL SELECT statement*
 - *Dynamically create columns in a table component*
 - *Based on which database columns were returned*
 - *Take column headers from metadata*
- *Demo of a SQL Browser in action ...*

- Well designed component models allow components to be *self describing*
- *Examples of component description:*
 - *(Localized) display name – to show in a palette*
 - *Tooltip text to be shown when a mouse is hovered over this component*
 - *Popup help text to be displayed about this component on demand*
 - *List of the properties of this component, to be included on a property sheet*
 - *For each property, the option to specify a Property Editor to use for customizing values*
 - *Design time behavior when the user interacts with this component:*
 - *When a new component is dropped on a page, create a set of child components*
 - *React to a property change event by modifying related state on other components*
- *The result is you can provide graphical Interactive Development Environment (IDE) tools like Java Studio Creator*
- *Demo of user component manipulation inside a tool ...*

Frameworks And AJAX

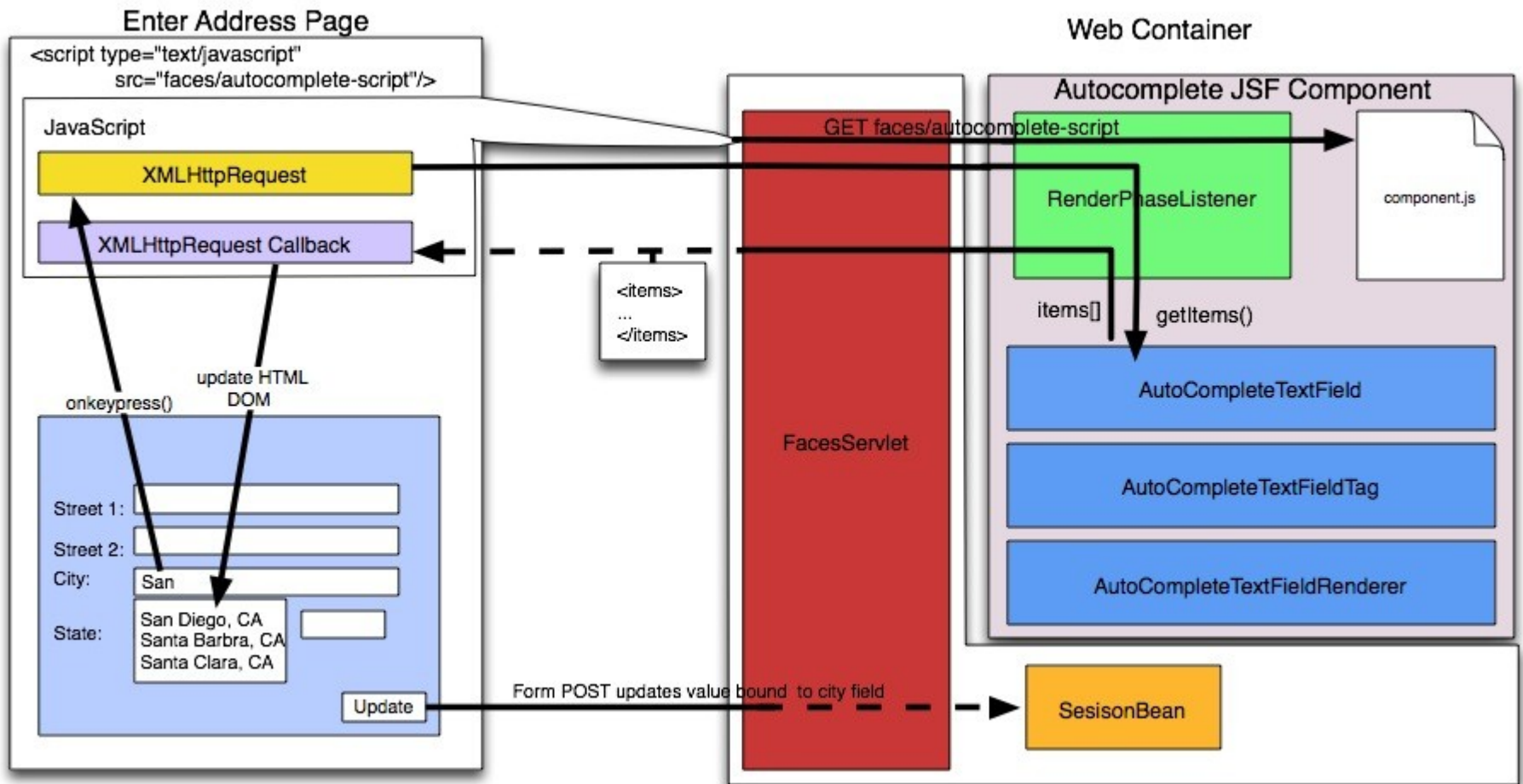
- It is almost impossible to ignore the hype around AJAX recently
- The techniques are not actually new:
 - XMLHttpRequest has been part of Internet Explorer for five years
 - Alternative techniques (such as hidden frames) have been available even longer
- What is new:
 - Reasonably portable XMLHttpRequest implementations in other browsers
 - Improvements in DHTML and JavaScript implementations in general
 - Increasing desire to combine two goals:
 - Easy deployment of a web-based application
 - Better user experience of a rich client application
- What does it mean for framework architectures?
 - One option is to simply ignore it

- Sequence diagram for a conceptual implementation
 - No framework involved beyond the Servlet API
 - But requires the developer to deal with JavaScript and DHTML directly



- Client-server integration services:
 - WebWork – client side validation with asynchronous server callbacks
 - WebWork – AJAX Theme dynamic partial page reloading
 - Shale Remoting – Serve static resources (javascript, stylesheets) asynchronously
 - Shale Remoting – Map asynchronous requests for dynamic data to business logic
- Encapsulate AJAX behavior in “widgets”:
 - WebWork – tabbed panel with content loaded asynchronously
 - Tapestry “Tacos” Components – modal dialog, tree control
 - Apache MyFaces – AJAX-enabled JavaScript components
 - BluePrints Catalog – AJAX-enabled JavaScript components
- Let us look at the architecture of our Auto Complete Text Field widget, when implemented as a JavaServer Faces Component

Auto Complete Text Field Component



- Demo of the Auto Complete Text Field component in action ...
- Benefits of wrapping AJAX functionality in “widgets”:
 - Isolates application developer from complex JavaScript and DHTML interactions
 - Allows developer to focus on model tier interactions
 - Familiar component oriented development paradigm (same as non-AJAX components)
 - Enables tools to support development of AJAX based applications

- Web application frameworks became popular because they:
 - Address usability limitations in the low level standard APIs
 - Encourage better application architecture by separating concerns
 - Provide structure to allow focus on individual application elements
- Web application frameworks address a common set of issues:
 - Modelling of page navigation decisions
 - Provisions for accessing model tier resources
 - Representation of static and dynamic markup
 - Mapping incoming requests to business logic
- Key distinguishing characteristics between frameworks:
 - Fundamental design pattern (typically *Front Controller* or *View Helper*)
 - *Support (or not) of a user interface component model*
- *It is possible to gain the benefits of both design patterns, and the benefits of components, with a framework like JavaServer Faces*

- Cocoon (web application framework)
 - <http://cocoon.apache.org/>
- Facelets (alternative view handler for JavaServer Faces)
 - <https://facelets.dev.java.net/>
- Java BluePrints Solutions Catalog
 - <https://bpcatalog.dev.java.net/>
- JavaServer Faces (standard component API and framework)
 - <http://java.sun.com/j2ee/javaxserverfaces/>
- JavaServer Pages
 - <http://java.sun.com/products/jsp/>
- Java Studio Creator (IDE for web application development)
 - <http://developers.sun.com/jscreator>

- Servlet API
 - <http://java.sun.com/products/servlet/>
- Shale (framework that extends JavaServer Faces)
 - <http://struts.apache.org/struts-shale/>
- Struts (web application framework)
 - <http://struts.apache.org/>
- Spring MVC (web application framework)
 - <http://www.springframework.org/>
- Tapestry (web application framework)
 - <http://jakarta.apache.org/tapestry/>
- WebWork (web application framework)
 - <http://www.opensymphony.com/webwork/>