The State Of Web Frameworks

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

jax 2006
Konferenz für Java™, XML, Web Services

Agenda

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

Background

- Web tier APIs among the first standardized
  - Servlet (1996)
  - JavaServer Pages (1999)
- But the standards stopped at foundations:
  - Low level abstraction of HTTP
  - Easy mechanisms for static/dynamic markup
- Resulted in much open source innovation

Servlet API – Foundation

- Abstracted the basic HTTP concepts:
  - Request, Response, Servlet
- Added a usability feature for statelessness
  - Session
- It is *possible* to write a pure servlet app:
  - writer.println(„<td>Customer:</td>“);
  - writer.println(„<td>“ + cust.getName() + „</td>“);

Servlet API – Issues

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

JSP 1.0 – Inside Out Servlets

- Even in dynamic apps, much static content
- Servlets embed both content types in code
- What if we could embed dynamic content generation in static markup?
- JSP 1.0 supported three flavors:
  - Variables (<%! String foo; %>)
  - Expressions (<%= foo %>)
  - Scriptlets (<% foo = cust.getLastName(); %>

JSP 1.1 – Reduce Embedded Java

- Embedded Java code still has issues:
  - Developers must be familiar with Java
  - Different syntax/semantics than JavaScript
  - Still intermixes presentation/business logic
- JSP 1.1 introduced Custom Tags:
  - Page author deals with markup elements
  - Java code abstracted into separate classes
- But JSP's reputation still based on scriptlets

Web Application Frameworks

- While standards evolved, innovative solutions were being explored:
  - Application architecture frameworks
  - User interface component models
- To meet specific needs:
  - „Hello world“ examples not good enough base
- How does an architect choose from among the 50 alternatives?

Variations On A Theme

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

Variations On A Theme

- Key architectural decisions:
  - 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 UI component model
- We will briefly review the first three
- The latter two merit a deeper look

Page Navigation Decisions

- **Destination** based navigation:
  - Source view knows the name or URL of the destination view
  - Example – Tapestry action listener can:
    - Navigate to a specific URL
    - Be injected with an IPage representing destination
  - Example – Spring MVC's *ModelAndView* return value

Page Navigation Decisions

- **Outcome** based navigation:
  - Source view returns a *logical outcome*
  - External configuration information defines transition rules
  - Example – Struts *Action* returns *ActionForward*
  - Example – JavaServer Faces action method returns outcome string

Accessing Model Tier Resources

- Most frameworks are agnostic here
- This is a mark of good architectural design:
  - Web application architecture should not dictate architecture of business logic or persistence strategy
  - Allows existing implementations to be reused
  - Encourages tier-specific unit tests

Representing Markup

- Most frameworks support JSP for this:
  - Static markup is entered inline
  - Dynamic markup entered with custom tags
- Tapestry is an interesting exception
- Some frameworks support other choices:
  - Templating systems
  - XML documents transformed by XSLT

Markup Example – JSF

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

Representing Markup

- Tapestry takes a different approach:
  - Entire page represented in (almost) pure HTML
  - Dynamic content identified by HTML elements with a **jwcid** attribute
  - When rendered, dynamic content replaces this element
- Can use standard HTML editor tools
- JSF can use plug-ins to do this (Facelets)

Markup Example – Tapestry

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

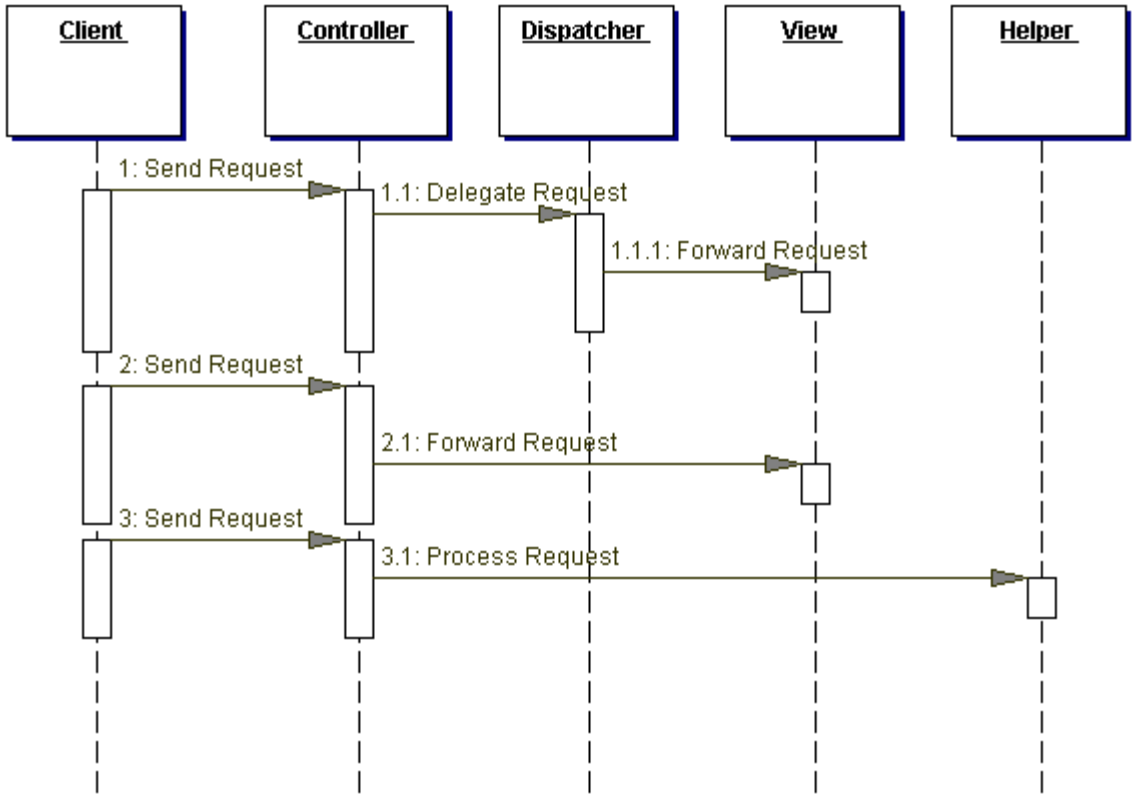
Design Patterns

- A common mechanism for understanding frameworks is to examine *design patterns* they implement
- A seminal book popularized this term:
  - Design Patterns: Elements of Reusable Object Oriented Software
- In the J2EE space, another book is useful:
  - Core J2EE Patterns: Best Practices & Strategies

Design Patterns

- Two J2EE patterns are popular in web application frameworks:
  - *Front controller*
  - *View helper*
- Let's briefly look at these patterns in a little more detail

The *Front Controller* Pattern



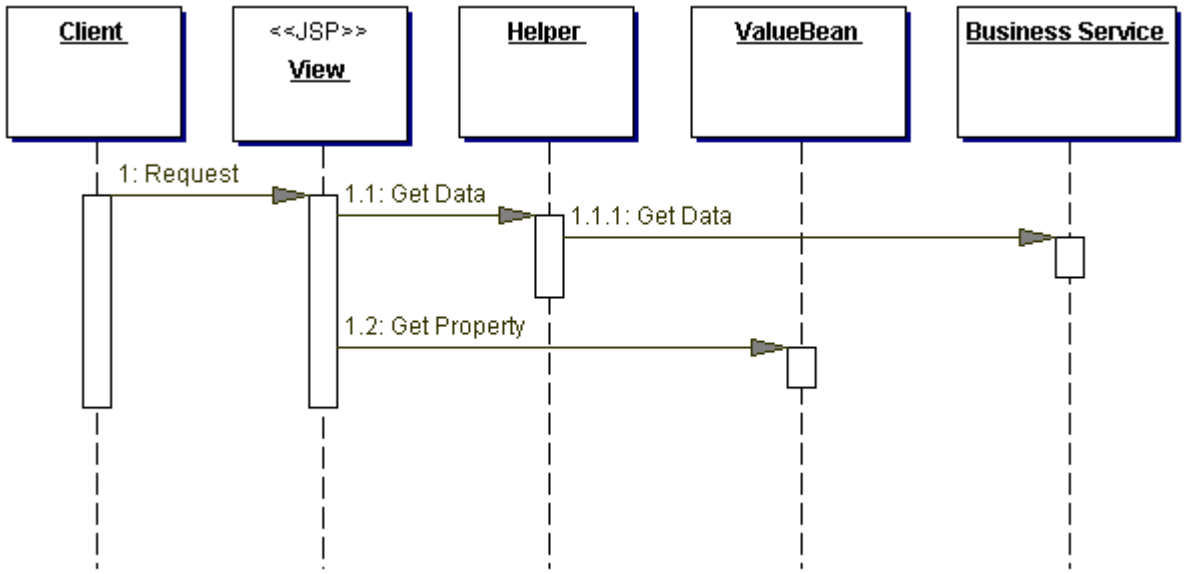
The *Front Controller* Pattern

- Participants and responsibilities:
  - *Controller* – initial contact point (may delegate)
  - *Dispatcher* – view management, navigation
  - *View* – display information to client
  - *Helper* – assistant to controller or view

The *Front Controller* Pattern

- Consequences of using this pattern:
  - *Centralizes control* – Single customization point
  - *Improves security management* – One way in
  - *Improves partitioning, reuse, maintainability* – Encourages clean separation of business logic and presentation logic

The *View Helper* Pattern



The *View Helper* Pattern

- Participants and responsibilities:
  - *View* – display information to client
  - *Helper* – assistant to view
  - *Value Bean* – specialized helper for state
  - *Business Service* – business logic to be accessed

The *View Helper* Pattern

- Consequences of using this pattern:
  - *Improves role separation* – Reduces dependencies between tiers
  - *Improves partitioning, reuse, maintainability* – Encourages clean separation of business logic and presentation logic

Can I Have It Both Ways?

- The two patterns share a consequence
- But they also feature unique benefits
- An ideal framework would allow a combo
- But how can we do that?
  - The development models look totally different

One Application–Two Approaches

- Struts has always included a canonical example *Mail Reader* application
- Let's examine two implementations:
  - *Front Controller* – Using Struts 1.2
  - *View Helper* – Using JSF 1.1 and Shale
- Both implementations available online:
  - http://struts.apache.org/

One Application–Two Approaches

- Both versions share a persistence tier
- Functionality of both versions is identical
  - Including client side JavaScript for validation
- Struts solution – complexity metrics:
  - 6 JSP pages, 10 Java classes
  - Complex configuration metadata
  - *Action* classes separate from form beans

One Application–Two Approaches

- JSF+Shale solution complexity metrics:
  - 6 JSP pages, 8 Java classes
    - No form beans
  - Straightforward configuration metadata
    - Managed bean per page
    - Navigation rules per page
  - Backing bean classes have instance properties for intermediate view state

One Application–Two Approaches

- Comparing the solution approaches:
  - Complexity of JSP pages roughly the same
    - Struts HTML tags, JSF UI component tags
  - Complexity of actions roughly the same
    - Pull data from request, call business logic
  - Overall *shape* of the application is very similar:
    - Design pattern of framework is mostly internal detail
    - Does not represent, by itself, a reason to choose one framework over another

Extending The Application

- What about adding a new feature?
  - Ensure user is logged on before page is accessed
- In a *front controller* framework like Struts:
  - Individual check in each JSP page
  - Container managed security or servlet filter
  - Customize controller to add logged in check

Extending The Application

- In a *view helper* framework like JSF:
  - Individual check in each JSP page
  - Container managed security or servlet filter
  - JSF *PhaseListener* to check on each request
  - Customize default *ActionListener* before invoking each action

Extending The Application

- What about adding a new feature?
  - Enforce common look and feel with banner
- In a *front controller* framework like Struts:
  - Hard code each page to match
  - Post-processing filter like SiteMesh
  - Integrated layout management with Tiles

Extending The Application

- In a *view helper* framework like JSF+Shale:
  - Hard code each page to match
  - Post-processing filter like SiteMesh
  - Integrated layout management with Tiles

Extending Application – Lessons

- The underlying architecture of the framework:
  - Is interesting (perhaps more so to framework geeks :-)
  - Influences which customizations are easy
  - Does not *a priori* help much in selection process
- Why else might I choose a framework?

Framework Differentiation

- The final *variation on a theme*:
  - Existence (or not) of a UI component model
- In a Swing app, there is no question:
  - Would **you** like to program to java.awt.Canvas?
- Web apps evolved from a world of handwritten markup:
  - Immature, non-standardized browsers
  - Limited or no development tools

Framework Differentiation

- Culture evolved that *page designer* was totally in charge of visual appearance:
  - Customized HTML development tools emerged
  - Tools did „code generation“ of markup
- 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

User Interface Components

- Some frameworks define UI components
- UI component responsibilities:
  - Rendering the appropriate visualization
  - Maintain state of user's interaction
  - Perform validation (correctness checks)
  - Convert input to correct model data types
  - Typically *bound* to model tier data

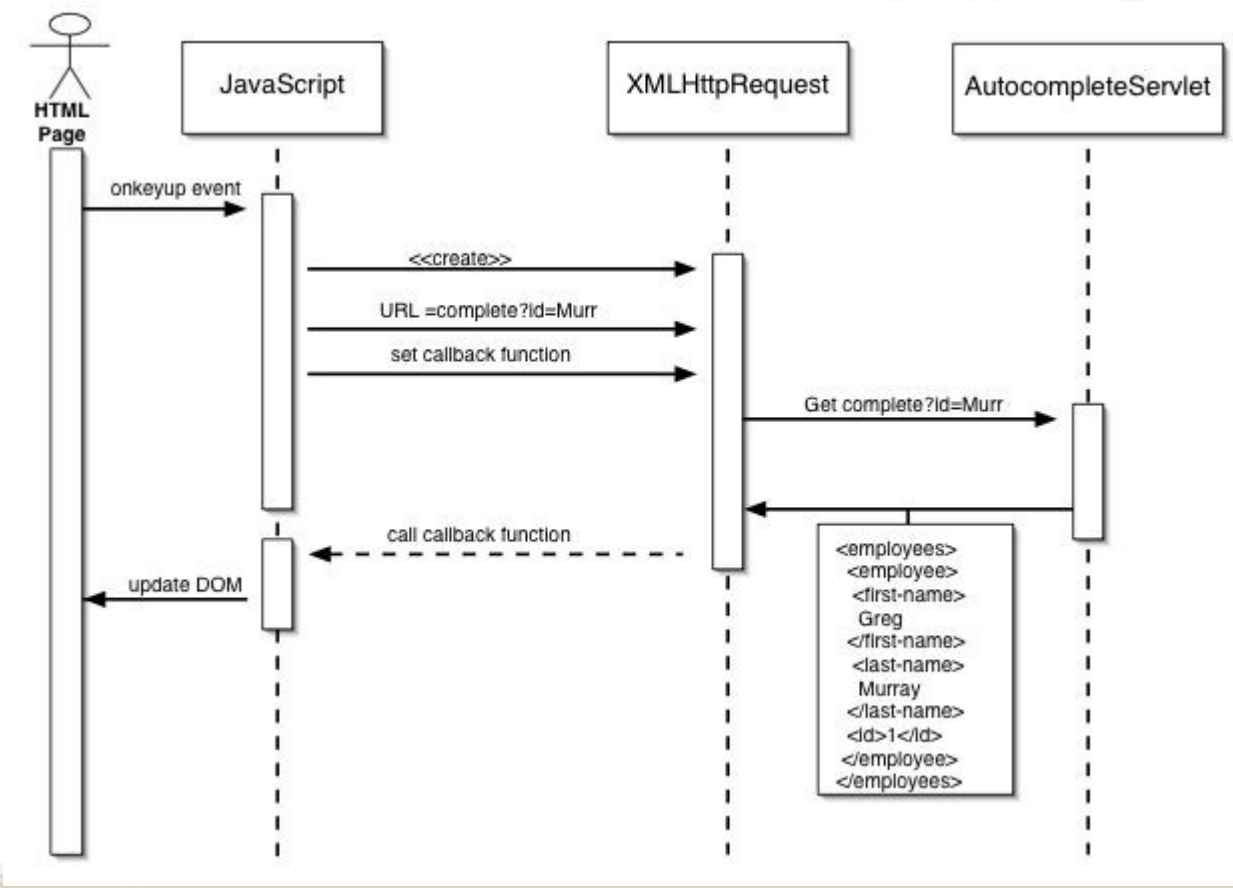
UI Component Model

- A UI component **model** also supports:
  - Hierarchical relationships between components
  - *Layout* components manage appearance
  - Dynamic modification of the hierarchy
- Example – DHTML lets you manipulate DOM dynamically on the client side
- Components can be self describing:
  - Create better experience inside a tool

Asynch XML and JavaScript

- Impossible to ignore hype around AJAX
- Techniques are not actually new
- What is new is a synergy:
  - Reasonably portable XMLHttpRequest APIs
  - Robust DHTML and JavaScript implementations
- Coupled with a desire for more interactive web UIs:
  - Without giving up deployment ease of webapps

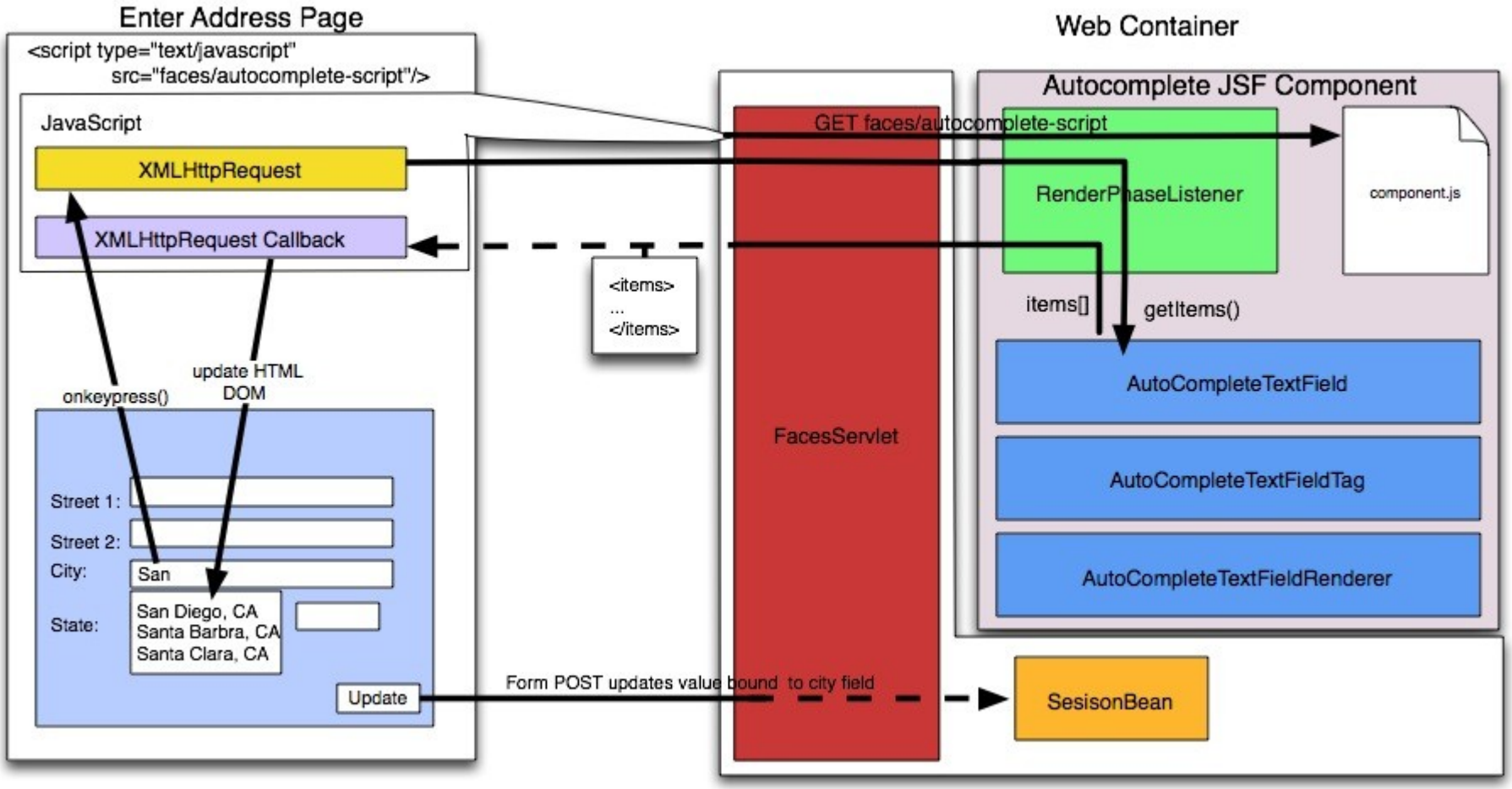
AutoComplete Text Field By Hand



Asynch XML and JavaScript

- Frameworks can offer value add services:
  - Client side validation
  - Partial page refresh
  - Serve static resources
  - Map dynamic requests to business logic
- Component frameworks can encapsulate AJAX behavior in „widgets“

AutoComplete JSF Component



AutoComplete JSF Component

- Benefits of encapsulating AJAX behavior:
  - Isolate developer from complex DHTML/JS
  - Allow developer to focus on model tier interactions
  - No new development paradigm to learn
    - AJAX and non-AJAX components used similarly
  - Component supporting tools can support development of AJAX based applications

Summary

- Web application frameworks became popular because they:
  - Addressed usability limitations of standard APIs
  - Encouraged better application architectures
  - Provided structure to focus on individual pieces

Summary

- 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

Summary

- Key distinguishing characteristics:
  - Fundamental design pattern used internally
    - Normally *front controller* or *view helper*
  - Support (or not) for UI component model
- It is possible to build a framework that provides the benefits of both patterns
- JavaServer Faces is one such framework

Thank You!