

# JS-based UI, Single Page Applications

Martin Ledvinka

[martin.ledvinka@fel.cvut.cz](mailto:martin.ledvinka@fel.cvut.cz)

Winter Term 2016



# Contents

- 1 Front-end Technologies - Historical Overview
  - Web Applications
  - Java World
- 2 JavaScript-based UI
  - Principles
- 3 Integrating JavaScript-based Frontend with Backend
- 4 Single Page Applications
  - Client Architecture



# Front-end Technologies - Historical Overview



# Web Applications

- <http://www.evolutionoftheweb.com/>



# Servlet API

- (HTTP-specific) classes for request/response processing,
- Response written directly into output stream sent to the client,
- Able to process requests concurrently.

```
public class ServletDemo extends HttpServlet{

    public void doGet (HttpServletRequest request, HttpServletResponse
        response)
        throws IOException{
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<h1>Hello World!</h1>");
        out.println("</body>");
        out.println("</html>");

    }
}
```



# Java Server Pages

- JSPs are text-based files containing:
  - Static data, usually HTML markup or XML,
  - JSP technology elements for creating dynamic content,
- JSPs are compiled into Servlets and returned to the client in response,
- JSP Standard Tag Library (JSTL) - a library of common functionalities – e.g. forEach, if, out.

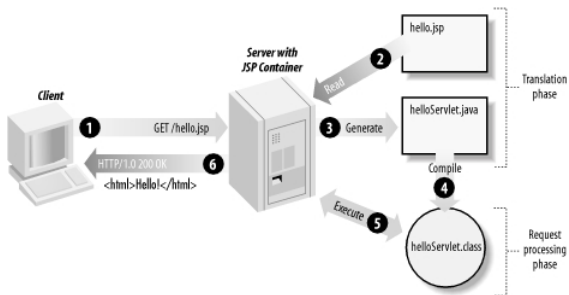


Figure: JSP processing. From

[http://www.onjava.com/2002/08/28/graphics/Jsp2\\_0303.gif](http://www.onjava.com/2002/08/28/graphics/Jsp2_0303.gif)

# JSP Example

```
<html>
<head>
  <title>JSP Example</title>
</head>
<body>
  <h3>Choose a hero:</h3>
  <form method="get">
    <input type="checkbox" name="hero" value="Master Chief">Master Chief
    <input type="checkbox" name="hero" value="Cortana">Cortana
    <input type="checkbox" name="hero" value="Thomas Lasky">Thomas Lasky
    <input type="submit" value="Query">
  </form>

  <%
String[] heroes = request.getParameterValues("hero");
if (heroes != null) {
%>
  <h3>You have selected hero(es):</h3>
  <ul>
  <%
    for (int i = 0; i < heroes.length; ++i) {
  %>
    <li><%= heroes[i] %></li>
  <%
    }
  %>
  </ul>
  <a href="<%= request.getRequestURI() %>">BACK</a>
  <%
  }
  %>
</body>
```



# Java Server Faces

- Component-based framework for server-side user interfaces,
- Two main parts:
  - An API for representing UI components and managing their state; handling events, server-side validation, and data conversion; defining page navigation; supporting internationalization and accessibility; and providing extensibility for all these features,
  - Custom JSP tag libraries for expressing UI components.
- Components make it easier to quickly develop complex applications,
- Many component libraries - PrimeFaces, RichFaces, IceFaces.





# JSF Lifecycle

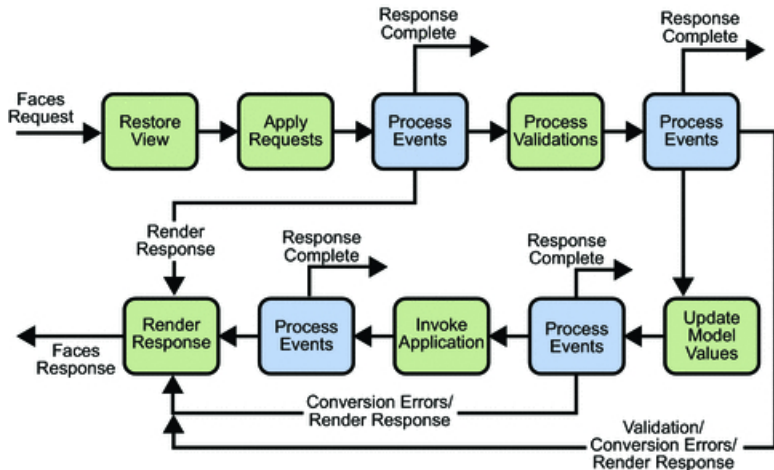


Figure: JSF lifecycle. From <http://docs.oracle.com/javaee/5/tutorial/doc/figures/jsfIntro-lifecycle.gif>



# JSF Example

```

<f:view>
  <h:head>
    <title>Book store – Users</title>
  </h:head>
  <h:body>
    <h1 class="title ui-widjet-header ui-corner-all"><h:outputText value="#{msg['user-list.title']}" /></h1>
    <p:panel>
      <h:form>
        <p:growl />
        <p:dataTable var="user" value="#{usersBack.users}">
          <p:column headerText="User">
            <p:commandLink action="#{selectedUser.setUserById('user')}" ajax="false">
              <h:outputText value="#{user.userName}" />
              <f:param name="userid" value="#{user.id}" />
            </p:commandLink>
          </p:column>
          <sec:ifAllGranted roles="ROLE_ADMIN">
            <p:column headerText="Delete User">
              <p:commandButton value="Delete" actionListener="#{usersBack.deleteUser(user.id)}"
                update="@form" />
            </p:column>
          </sec:ifAllGranted>
          <p:column headerText="Age">
            <h:outputText value="#{user.age}" />
          </p:column>
        </p:dataTable>
        <p:link outcome="book-store-welcome-page" value="Home" />
      </h:form>
    </p:panel>
    <p:commandLink action="#{loginBean.logout()}" value="Logout" />
  </h:body>
</f:view>

```



## JSF Example II

```
@Component("usersBack")
@Scope("session")
public class UsersBack {

    @Autowired
    private UserService userService;

    public List<UserDto> getUsers() {
        return userService.findAllAsDto();
    }

    public void deleteUser(Long userId) {
        userService.removeById(userId);
        FacesContext.getCurrentInstance().addMessage(null, new FacesMessage("User was successfully deleted"));
    }
}
```



## Other Popular Frameworks

**Google Web Toolkit (GWT)** Write components in Java, GWT then generates JavaScript from them,

**Vaadin** Built on top of GWT,

**Wicket** Pages represented by Java class instances on server.



# Why are we moving away from JSF?

- JSP, JSF are based on request/response, which requires frequent page reloads,
- Very limited support for AJAX,
- Limited support for mobile devices,
- Difficult to add new components.



# JavaScript-based UI



# JavaScript-based UI

- Client-side interface generated completely or partially by JavaScript,
- Based on AJAX,
  - Dealing with asynchronous processing,
  - Events – user, server communication,
  - Callbacks,
  - When done wrong, it is very hard to trace the state of the application,
- Enables dynamic and fluid user experience.

## No jQuery

- We will not be using jQuery,
- It is a collection of functions and utilities for dynamic page manipulation/rendering,
- But building a complex web application solely in jQuery is difficult and the code easily becomes messy.

# JS-based UI Classification

## Declarative

HTML templates with bindings for JS framework. E.g. Angular.

```
<html ng-app="appname">
  <head>
    <script src="js/angular.min.js"></script>
    <link href="style.css" rel="stylesheet"/>
    <script src="js/script.js"></script>
  </head>
  <body>
    <div ng-controller="appCtrl">
      <p>{{greeting.text}}, world </p>
    </div>
  </body>
</html>
```





# JS-based UI Classification

## Procedural

DOM tree is completely generated by JS. E.g. ReactJS.

```
var HelloMessage = React.createClass({
  render: function () {
    return <h1>Hello {this.props.message}!</h1>;
  }
});

ReactDOM.render(<HelloMessage message="World" />, document.getElementById('root'));
```



# JS-based UI Principles

- Application mostly responds by manipulating the DOM tree of the page,
- Fewer refreshes/page reloads,
- Server communication happens in the background,
- Single-threaded (usually),
- Asynchronous processing.



# Integrating JavaScript-based Frontend with Backend



# Frontend – Backend Communication

- JS-based frontend communicates with REST web services of the backend,
- Usually using JSON as data format,
- Asynchronous nature,
  - Send request,
  - Continue processing other things,
  - Invoke callback when response received.



# Frontend – Backend Communication Example

```

'use strict';

var Reflux = require('reflux');

var Actions = require('../actions/Actions');
var Ajax = require('../utils/Ajax');

var SearchStore = Reflux.createStore({
  init: function () {
    this.listenTo(Actions.fullTextSearch, this.onFullTextSearch);
  },

  onFullTextSearch: function (expr) {
    Ajax.get('rest/search?expression=' + encodeURIComponent(expr)).end((data) => {
      this.trigger({
        action: Actions.fullTextSearch,
        data: data
      });
    });
  }
});

module.exports = SearchStore;

```



```

GET /inbas-reporting-tool-refactoring/rest/search?expression=drahy HTTP/1.1
Host: www.inbas.cz
Connection: keep-alive
Accept: application/json
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
/54.0.2840.71 Safari/537.36

```



# Frontend – Backend Communication Example II

```

@RestController
@RequestMapping("/search")
public class SearchController {

    static final String EXPRESSION_PARAM = "expression";

    @Autowired
    private SearchService searchService;

    /**
     * Runs a full-text search for the specified expression.
     *
     * @param expression The expression to search for
     * @return Search results
     */
    @RequestMapping(method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
    public RawJson fullTextSearch(@RequestParam(value = EXPRESSION_PARAM, defaultValue = "") String expression) {
        if (expression.isEmpty()) {
            throw new BadRequestException("Cannot search for an empty string.");
        }
        return searchService.fullTextSearch(expression);
    }
}

```



```

HTTP/1.1 200 OK
Date: Sat, 29 Oct 2016 16:44:15 GMT
Server: Apache/2.4.10 (Debian)
Content-Type: application/json

```

```

{
  // JSON response body
}

```



# Frontend – Backend Communication Example III



```

'use strict';

var Reflux = require('reflux');

var Actions = require('../actions/Actions');
var Ajax = require('../utils/Ajax');

var SearchStore = Reflux.createStore({
  init: function () {
    this.listenTo(Actions.fullTextSearch, this.onFullTextSearch);
  },

  onFullTextSearch: function (expr) {
    Ajax.get('rest/search?expression=' + encodeURIComponent(expr)).end((data) => {
      this.trigger({
        action: Actions.fullTextSearch,
        data: data
      });
    });
  }
});

module.exports = SearchStore;

```



# Single Page Applications





# Single vs. Multi Page JS-based Web Applications

**Multi Page Web Applications** Individual pages use a lot of JS, but browser navigation still occurs – browser URL changes and page reloads. Example: GitHub.

**Single Page Web Applications** No browser navigation occurs, everything happens in one page using DOM manipulation. Example: Gmail.











# Single Page Applications

- Provide more fluid user experience,
- No page reloads, only URL hash usually changes,
- View changes by modifications of the DOM tree,
- Most of the work happens on the client side,
- Communication with the server in the background,
- Client architecture becomes important – a lot of code on the client.



# Single Page Application Specifics

- Everything has to be loaded when page opens,
  - Framework,
  - Application bundle,
  - Most of CSS,
- Different handling of security,
- Different way of navigation,
- Difficult support for bookmarking.

 bootstrap.min.css	200	stylesheet	<a href="#">j.spring_security.check-Infinity</a>	20.2 KB	28 ms
 bootstrap-datetimepicker.min.css	200	stylesheet	<a href="#">j.spring_security.check-Infinity</a>	1.5 KB	17 ms
 dhtmlxgantt.css	200	stylesheet	<a href="#">j.spring_security.check-Infinity</a>	9.8 KB	25 ms
 inbas-audit.min.css	200	stylesheet	<a href="#">j.spring_security.check-Infinity</a>	3.0 KB	21 ms
 dhtmlxgantt.js	200	script	<a href="#">j.spring_security.check-Infinity</a>	44.3 KB	63 ms
 dhtmlxgantt_tooltip.js	200	script	<a href="#">j.spring_security.check-Infinity</a>	1.9 KB	34 ms
 cs.js	200	script	<a href="#">j.spring_security.check-Infinity</a>	1.6 KB	39 ms
 bundle.min.js	200	script	<a href="#">j.spring_security.check-Infinity</a>	282 KB	166 ms



# Single Page Application Drawbacks

- Navigation and *Back* support,
- Scroll history position,
- Event cancelling (navigation),
- Bookmarking,
- SEO,
- Automated UI testing.



# Client Architecture

- JS-based clients are becoming more and more complex,
  - → necessary to structure them properly,
- Plus the asynchronous nature of AJAX,
- Several ways of structuring the client.

## Model View Controller (MVC)

- Classical pattern applicable in client-side JS, too,
- Controller to control user interaction and navigation, **no business logic**,
- Frameworks often support MVC.



## Client Architecture II

### Model View View-Model (MVVM)

- Originally developed for event-driven programming in WPF and Silverlight,
- View-Model is an abstraction of the View,
- Let the framework bind UI components to View-Model attributes (two-way binding),
- Controllers still may be useful.

### Flux

- Unidirectional flow,
- Originated in ReactJS,
- Simplifies reasoning about application state.



# The End

# Thank You



# Resources

- M. Fowler: Patterns of Enterprise Application Architecture,
- <https://dzone.com/articles/java-origins-angular-js>,
- <https://msdn.microsoft.com/en-us/magazine/dn463786.aspx>,
- <http://singlepageappbook.com/index.html>,
- <http://adamsilver.io/articles/the-disadvantages-of-single-page-applications/>,
- <http://www.oracle.com/technetwork/articles/java/webapps-1-138794.html>.

