Web Application Clients

Petr Aubrecht (Martin Ledvinka)

petr@aubrech.net

Winter Term 2020

Contents

Historical Overview

Web Applications

- http://www.evolutionoftheweb.com/
- + Mozilla

Frontend Technologies

Servlet API

- Fast API, faster than CGI used at the time (later FastCGI)
- (HTTP-specific) classes for request/response processing
- Response written directly into output stream sent to the client
- Processes requests concurrently, Servlet 3.0 with asynchronous calls
- Still used for non-HTML content (images, graphs, PDF)

```
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

- HTML or XML markup with pieces of Java code
- JSPs are compiled into Servlets, e.g. as fast as Servlets
- JSP Standard Tag Library (JSTL) - a library of common functionalities – e.g. `forEach`, `if`, `out`
- Combobox updating is a nightmare :-)

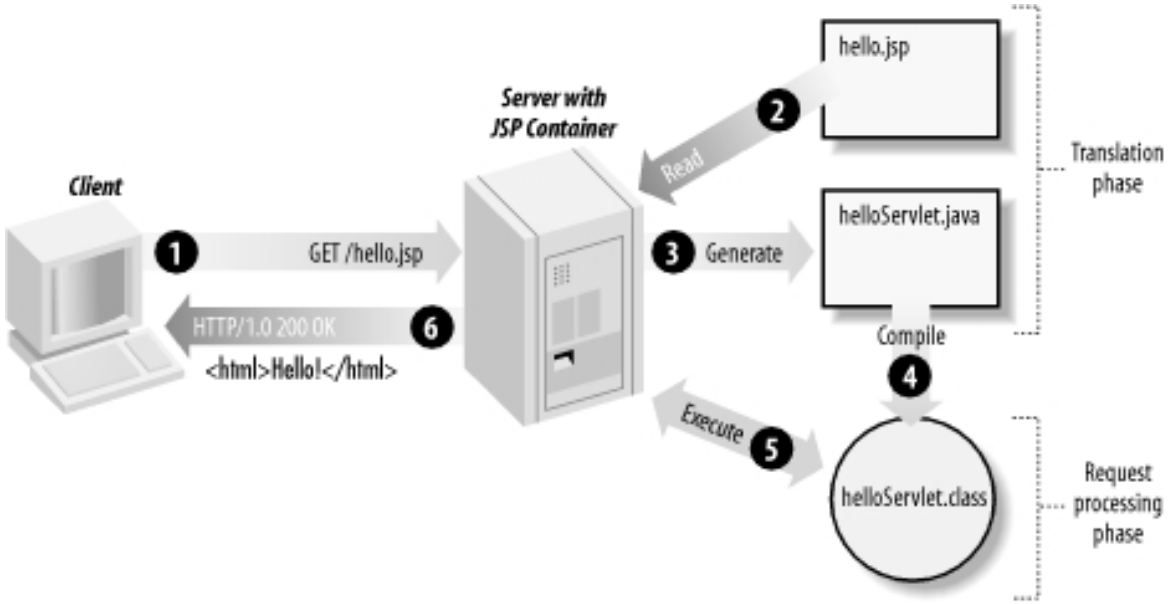


Figure : JSP processing. From `http://www.onjava.com/2002/08/28/graphics/Jsp2_0303.gif`

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
- XML based description of page, setting up components
- Expression language used to join to Java code
- Rich components make it easy to quickly develop typical information systems – PrimeFaces (!), RichFaces, IceFaces
- Component libraries add support for Ajax, templates
- Good choice for Java developers, most of functionality is done on server, easy connection between UI components and Java

JSF Example

```
<f:view>
    <h:head>
        <title>Book store – Users</title>
    </h:head>
    <h:body>
        <h1 class="title ui-widget-header ui-corner-all"><h:outputText value="#{msg['user-list.title']}"/></h1>
        <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>
                <p:column headerText="Delete User" render="#{security.admin}">
                    <p:commandButton value="Delete" action="#{usersBack.deleteUser}" update="@form" />
                </p:column>
                <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: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 sucessfully deleted."));
    }
}
```

JSF Lifecycle

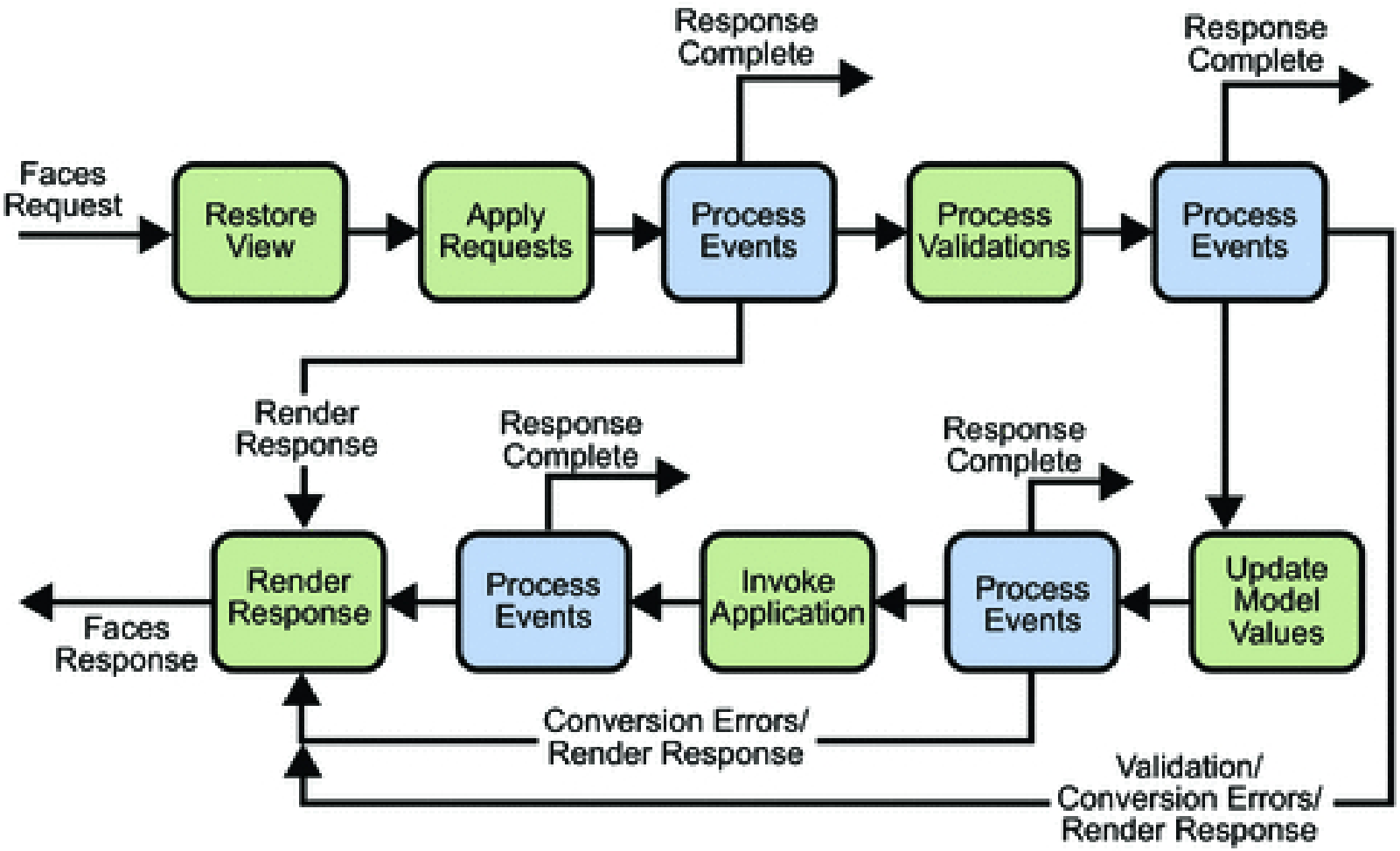


Figure : JSF lifecycle. From `http://docs.oracle.com/javaee/5/tutorial/doc/figures/jsfIntro-lifecycle.gif`

Features of Java Based UI

- Servlet – low-level, fastest
- JSP – simple interactive HTML page, like PHP, very fast
- JSF is based on request/response, which makes server request for every action.
- Page rendering (full or part of screen) happens on server.
  - Performance for heavy sites can be an issue
  - Not appropriate for apps like Google Office (lots of UI actions with rare communication to server)
- JSF offers rich components libraries for typical scenarios, e.g. tables with filters, sorting, paging, loading on-demand etc.
- Impossible to write offline apps.
- Stable technology – compatible for years
- Difficult to add new or significantly extend existing components, easy to make compound components.

Other Popular Frameworks

Google Web Toolkit (GWT) Write components in Java, GWT then generates JavaScript, can make fat client, client and server share Java objects, quite slow compilation due to compile for different browsers

Vaadin Built on top of GWT, no need to pre-compile Java→JS

Wicket Pages represented by Java class instances on server

Spring MVC Servlet-based API with various UI technologies, originally JSP, but also Thymeleaf, FreeMarker, Groovy Markup

JS-based UI Principles

- Application responds by manipulating the DOM tree of the page
- Fewer refreshes/page reloads, much more REST communication instead
- Server communication happens in the background
- Single-threaded
- Asynchronous processing

JavaScript-based UI

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

No jQuery

- Plain jQuery use discouraged
- 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, e.g. Angular.

```
<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>

<hero-detail *ngIf="selectedHero" [hero]="selectedHero"></hero-detail>
```

JS-based UI Classification

"Procedural"

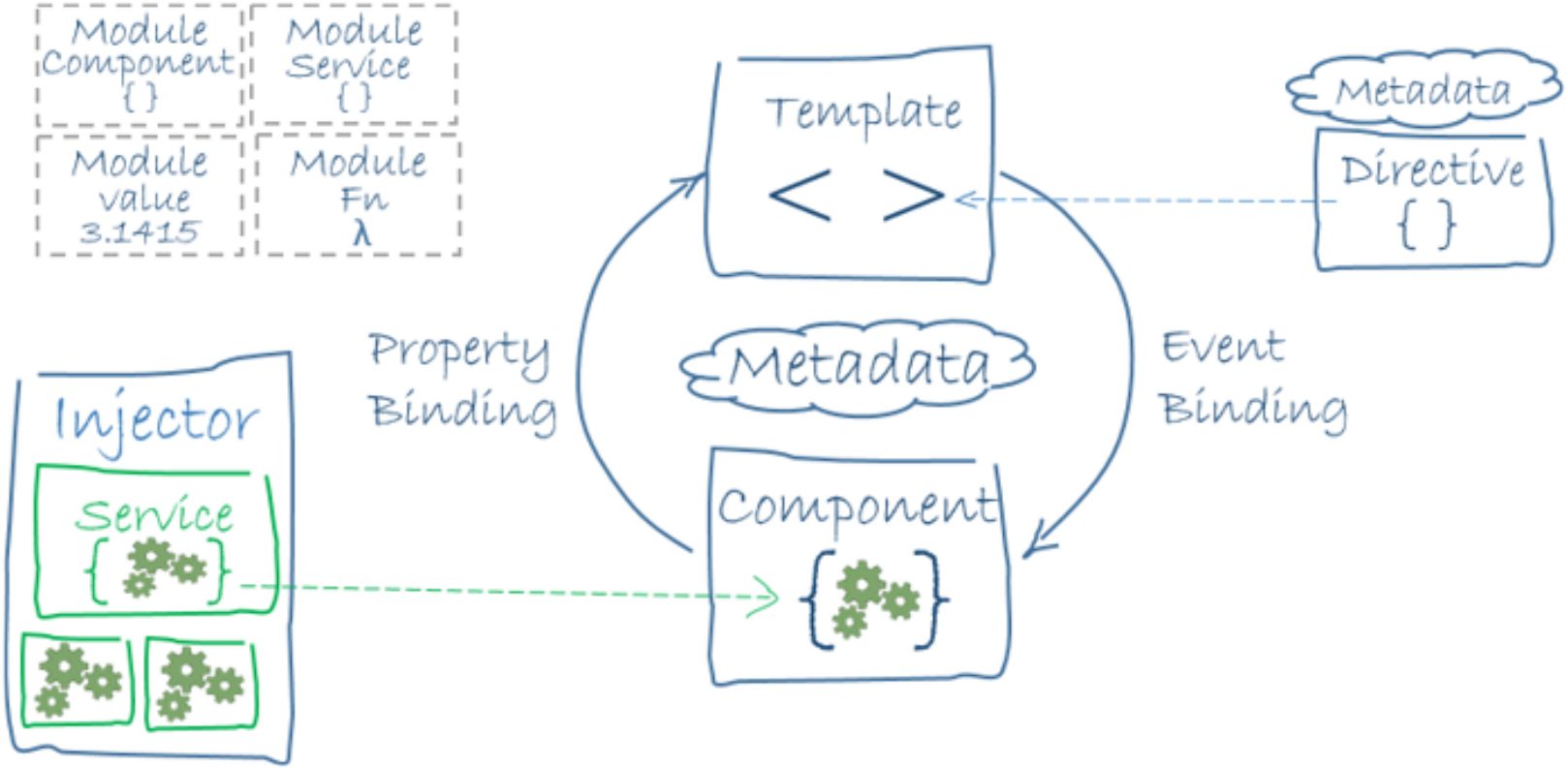
View structure is defined as part of the JS code, e.g. React.

```
class HelloMessage extends React.Component {
  render() {
    return <h1>Hello {this.props.message}!</h1>;
  }
}

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

Angular

- Developed by Google, current version – Angular 8.2.x
- Encourages use of MVC with two-way binding
- HTML templates enhanced with hooks for the JS controllers
- Built-in routing, AJAX
- `https://angular.io/`



Angular Example

```
import { Component } from '@angular/core';
import { Hero } from '../hero';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent {
  hero: Hero = {
    id: 117,
    name: 'Master Chief'
  };

  constructor() { }
}
```

```
<h2>{{hero.name}} Details</h2>
<div><span>id: </span>{{hero.id}}</div>
<div><span>name: </span>{{hero.name}}</div>
```

React

A JavaScript library for building user interfaces.

- Created and developed by Facebook
- Used by Facebook and Instagram
- React Native for developing native applications for iOS, Android and UWP in JS
- Easy to integrate into legacy web applications
- Let's take a look at the EAR e-shop UI
- `https://facebook.github.io/react/`

Other JS-based Alternatives

Ember

- Open source framework
- Templates using Handlebars
- Encourages MVC with two-way binding
- New components created using Handlebars templates + JS
- Built-in routing, AJAX
- http://emberjs.com/

```
1  <div>
2   <label>Name:</label>
3   {{input type="text" value=name placeholder="Enter your name"}}
4  </div>
5  <div class="text">
6   <h3>My name is {{name}} and I want to learn Ember!</h3>
7  </div>
```

Other JS-based Alternatives

BackboneJS

- Open source framework
- Provides models with key-value bindings, collections
- Views with declarative event handling
- View rendering provided by third-party libraries - e.g., jQuery, React
- Built-in routing, AJAX
- `http://backbonejs.org/`

```
var Todo = Backbone.Model.extend({


  defaults: function() {
    return {
      title: "empty todo...",
      order: Todos.nextOrder(),
      done: false
    };
  },


  toggle: function() {
    this.save({done: !this.get("done")});
  }

});
```

And many others...

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/resolve Promise when response received

Frontend – Backend Communication Example

```
export function loadCategories() {
    const action = {
        type: ActionType.LOAD_CATEGORIES
    };
    return (dispatch) => {
        dispatch(asyncActionRequest(action));
        return axios.get('rest/categories')
            .then(resp => dispatch(loadCategoriesSuccess(resp.data)))
            .catch(error => {
                if (error.response.data.message) {
                    dispatch(publishMessage({message: error.response.data.message, type: 'danger'}));
                }
                return dispatch(asyncActionFailure(action, error.response.data));
            });
    };
}
```

↓

```
GET /eshop/rest/categories HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Accept: application/json
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
      /61.0.3163.91 Safari/537.36
```

Frontend – Backend Communication Example II

```java
@RestController
@RequestMapping("/categories")
public class CategoryController {

    private static final Logger LOG = LoggerFactory.getLogger(CategoryController.class);

    private final CategoryService service;

    private final ProductService productService;

    @Autowired
    public CategoryController(CategoryService service, ProductService productService) {
        this.service = service;
        this.productService = productService;
    }

    @RequestMapping(method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
    public List<Category> getCategories() {
        return service.findAll();
    }
```

↓

```
HTTP/1.1 200 OK
Date: Sun, 17 Nov 2019 16:12:46 GMT
Server: Apache/2.4.10 (Debian)
Content-Type: application/json

{
  // JSON response body
}
```

Frontend – Backend Communication Example III

↓

```
export function loadCategories() {
    const action = {
        type: ActionType.LOAD_CATEGORIES
    };
    return (dispatch) => {
        dispatch(asyncActionRequest(action));
        return axios.get('rest/categories')
            .then(resp => dispatch(loadCategoriesSuccess(resp.data)))
            .catch(error => {
                if (error.response.data.message) {
                    dispatch(publishMessage({message: error.response.data.message, type: 'danger'}))
                }
                return dispatch(asyncActionFailure(action, error.response.data));
            });
    };
}
```

Single Page Applications

Single vs. Multi Page JS-based Web Applications

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

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

Single Page Applications

- Provide more fluid user experience
- No page reloads
- 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

- Almost 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	j_spring_security_check:-Infinity	20.2 KB	28 ms
bootstrap-datetimepicker.min.css	200	stylesheet	j_spring_security_check:-Infinity	1.5 KB	17 ms
dhtmlxgantt.css	200	stylesheet	j_spring_security_check:-Infinity	9.8 KB	25 ms
inbas-audit.min.css	200	stylesheet	j_spring_security_check:-Infinity	3.0 KB	21 ms
dhtmlxgantt.js	200	script	j_spring_security_check:-Infinity	44.3 KB	63 ms
dhtmlxgantt_tooltip.js	200	script	j_spring_security_check:-Infinity	1.9 KB	34 ms
cs.js	200	script	j_spring_security_check:-Infinity	1.6 KB	39 ms
bundle.min.js	200	script	j_spring_security_check:-Infinity	282 KB	166 ms

Single Page Application Drawbacks

- Navigation and *Back* support
- Scroll history position
- Event cancelling (navigation)
- Bookmarking
- SEO

Client Architecture

- JS-based clients are becoming more and more complex
  - $\rightarrow$ neccessary 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)

- Motivation – model cannot be simply presented in View, needs some conversion.
- **Models** hold application data. They're usually structs or simple classes.
- **Views** display visual elements and controls on the screen. They're typically subclasses of UIView.
- **View Models** transform model information into values that can be displayed on a view. They're usually classes, so they can be passed around as references.
- View controllers provides functionality of UI, owns both View and View models.

UI – Bits from The Last Year

- REST + React.js (Vue.js) are the most popular
- Flux/Redux are frequently used, useful for complex pages
- Raise of standard Web Components, JS starts providing functionality of e.g. React
- WebAssembly – possibility to run code other than JS, can run C code
- HTML 5 is now common – supports several codecs for video, still doesn't know combobox (edit with dropdown list)
- Angular is easy → Angular is hell, React is easy → React is hell, Vue is easy → ???
- Java is bad for its types → JS is better → Well, we have troubles in bigger projects → Move to TypeScript, types are cool → ???
- Server side is bad due to performance → Move to client-side rendering → JS is slow → Move to server-side rendering in JS → ???

JS-based UI – My Experience I

- JS is most frequently used language. And most hated ever.
- Revolution happens every half a year, no stable best practices (opinions change frequently)
- Frameworks change quickly, it is necessary rewrite applications continuously (e.g. spending money without any added value)
- JS is no more "write&run", it needs compile, often transpilation, etc.
- JS in UI effectively require JS on server, modern frameworks work badly in production mode.
- Reason for JS was performance, which is now returning back to server (server-side rendering)...
- JS leads to splitting teams to backend and frontend
- Appropriate for sites like Facebook.
- JSF accesses Java objects directly, JS requires every data exchange visible via REST, much more work and vectors for attack
- Duplicate validation on client & server

JS-based UI – My Experience II, Technologies

- **GWT** – perfect for Java programmers, full type-check, all Java, sustaining mode (Vaadin is still developped). It was great for fat clients for Java team.
- **JSF** – Java programmers learnt it quickly, easy to provide rich functionality, PrimeFaces actively developped, modern Features available (asynchronous processing, WebSocket), rarely strange behavior (e.g. methods with parameters called from table). Great for typical information systems.
- **JS, React.js** – basics are simple, with complexity rapidly grow problems like compilation sometimes suddenly breaks; strange behavior of this.props.router.query; when multiple REST calls are needed, it is difficult to render screen with partial data; complex correct handling of errors. The only solution for really fat client.
- Fullstack – GWT, JSF or JS + node.js allow fullstack developer, JS + REST needs two teams

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.