Web Application Clients

Petr Aubrecht (Martin Ledvinka)

petr@aubrecht.net

Winter Term 2022

Contents

1. Historical Overview
2. Java World
3. JavaScript-based UI
4. Single Page Applications
5. Frameworks
6. Integrating JavaScript-based Frontend with Backend
7. Client Architecture
8. Conclusion

Petr Aubrecht's View

- UI is ALWAYS difficult (especially for Java developer), mostly hated...
  - (X)HTML
  - CSS
  - JS
  - sometimes XSL-T + XSL-FO (PDFs)
- Note, how often is Facebook broken (now – image cannot be closed sometimes)
- P.A. joke: JS solves lots of problems, which don't exist without JS
- **...but it is the best place for innovation!**

Historical Overview

Evolution of the Web

- Offline backup of evolutionoftheweb.com

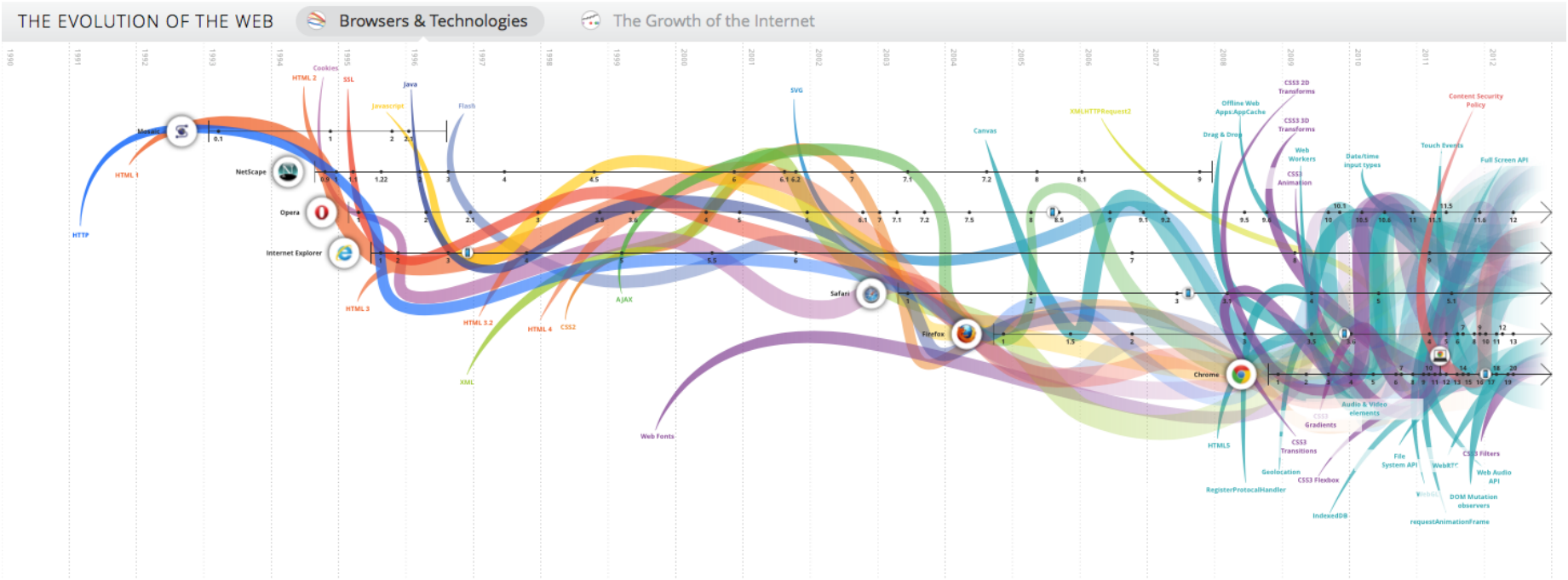


Figure: From `http://www.evolutionoftheweb.com/`

Web Applications

- `http://www.evolutionoftheweb.com/`
- + Mozilla 1998 after release Netscape source codes.
- 2022: several video codecs, no combobox, only JavaScript

Common Gateway Interface (CGI)

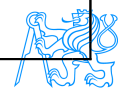
- Mid-1990s
- Server (httpd, Apache) starts a program
  - passes parameters in environment variables
  - stdout is returned to server and client
- Yes, it starts a program for every request
- Yes, written in C, Perl, later PHP
- No connection pools, no threads, no caching
- FastCGI – keep processes in memory

Java World

Servlet API

- Fast API, faster than CGI used at the time, May 1996
- (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 – simple!
- 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.

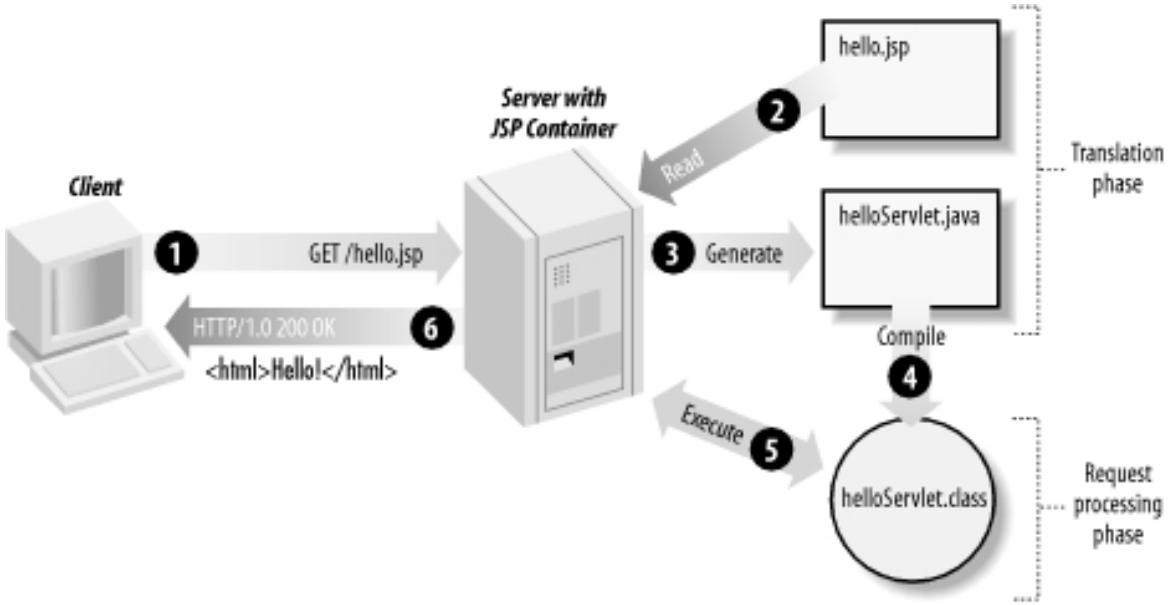


Figure: JSP processing. From `http://www.onjava.com/2002/08/28/graphics/Jsp2_0303.gif`

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

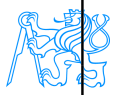
- Component-based framework for server-side user interfaces
- XML based description of page, setting up components
- Expression language used to join to Java code
- Rich components make it easy to quickly develop typical information systems – PrimeFaces (!), RichFaces, IceFaces
- Component libraries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

JSF Example I – Java

```
@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 Example II – XHTML

```
<h:body>
  <h1 class="title"><h:outputText value="#{msg['list.title']}"/></h1>
  <h:form>
    <p:dataTable var="user" value="#{usersBack.products}">
      <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"/>
    <p:commandLink action="#{loginBean.logout()}" value="Logout" />
  </h:form>
</h:body>
```

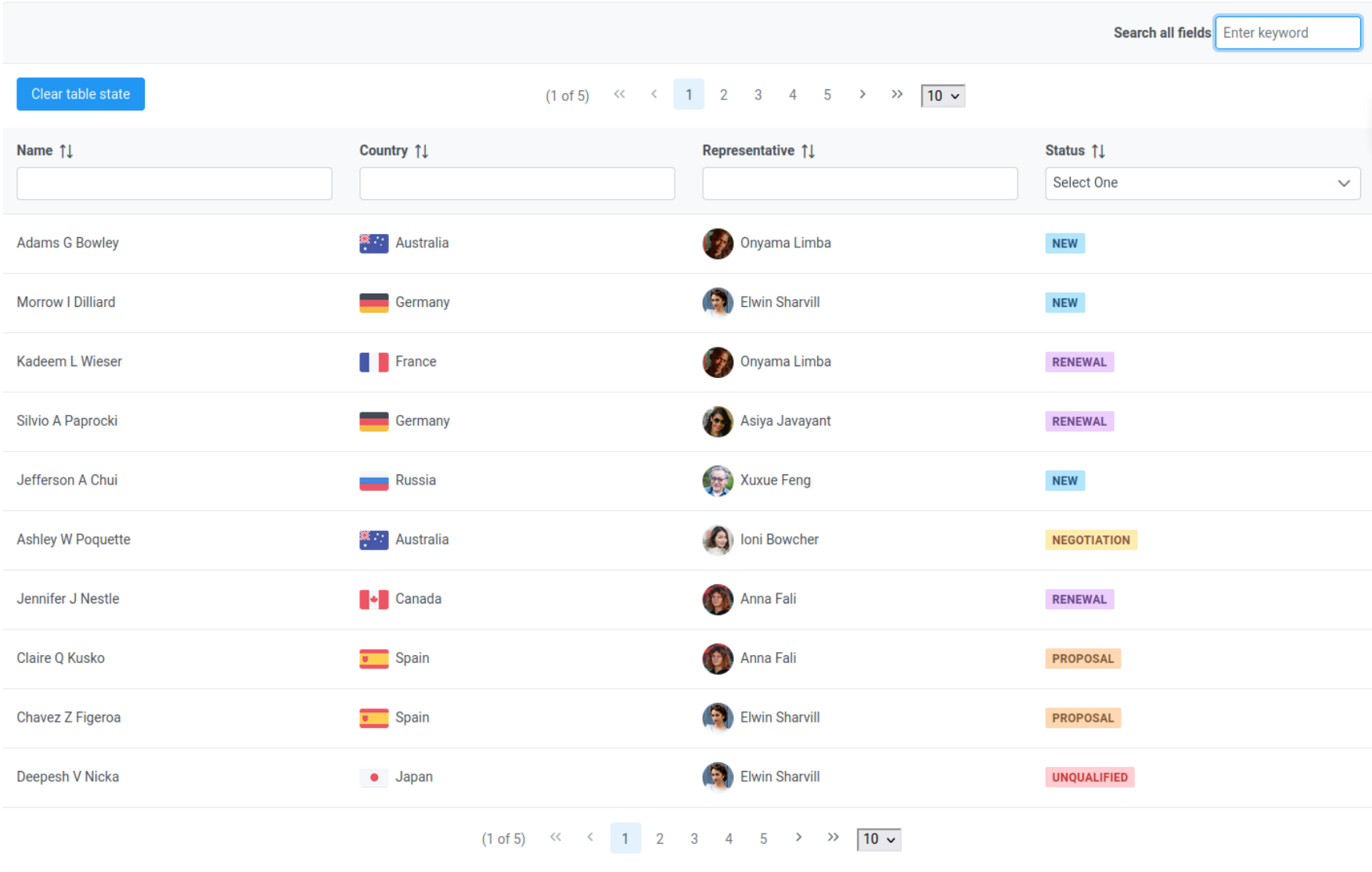


JSF Example III – Output

Code	Name	Category	Quantity
f230fh0g3	Bamboo Watch	Accessories	24
nvklal433	Black Watch	Accessories	61
zz21cz3c1	Blue Band	Fitness	2
244wgerg2	Blue T-Shirt	Clothing	25
h456wer53	Bracelet	Accessories	73
av2231fwg	Brown Purse	Accessories	0
bib36pfvm	Chakra Bracelet	Accessories	5
mbvjkgip5	Galaxy Earrings	Accessories	23
vbb124btr	Game Controller	Electronics	2
cm230f032	Gaming Set	Electronics	63

Figure: Format of the output, based on schema

JSF Example IV – PrimeFaces More Complete Output



JSF Lifecycle

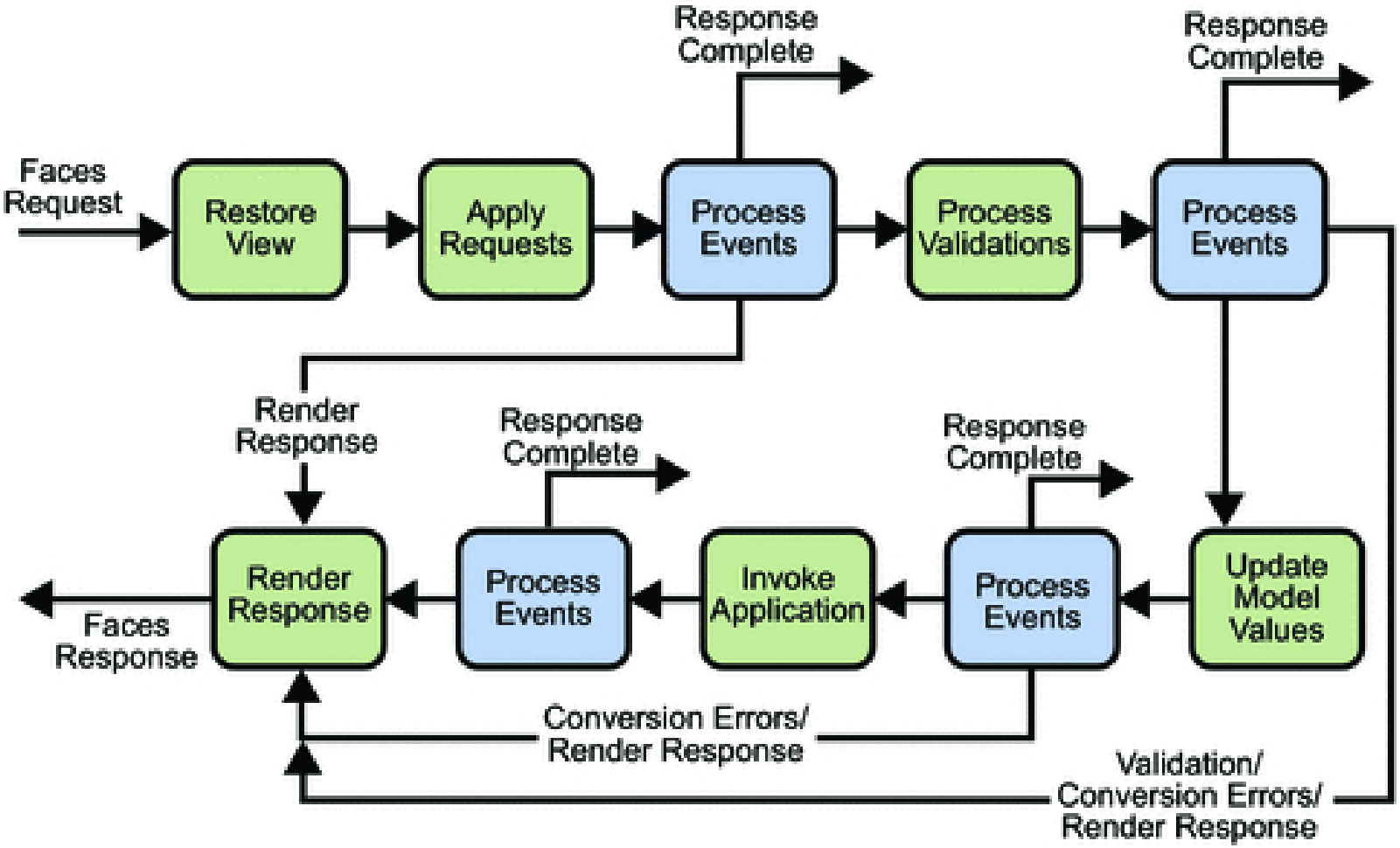


Figure: JSF lifecycle. From `http://docs.oracle.com/javaee/5/tutorial/doc/figures/jsfIntro-lifecycle.gif`

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 Originally built on top of GWT, no need to pre-compile Java→JS. Today, viable independent project.

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

JavaScript-based UI

JS-based UI Principles

- Application responds by manipulating the DOM tree of the page
- Fewer refreshes/page reloads, much more API communication instead
- Server communication happens in the background
- Single-threaded
- Asynchronous processing
- All of the expect NodeJS on server, otherwise they have big problems (e.g. page not found when refresh)
- All need compilation, mostly transpilation from a bit more sane language (TypeScript)
- Some parts run on both client and server.
- "Best practice" changes every ~~6~~ 3 months (jQuery, transpilation, Redux, classes vs hooks, Angular 2.0, etc.)
- Incompatible implementations – yes, still.

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

- jQuery is discouraged nowadays
- 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'));
```

Single Page Applications

Single Page Applications

- View changes by modifications of the DOM tree
- Use router – URL parameters (bookmarkable!) and internal state define content
- (Promise to) Provide more fluid user experience
- No page reloads (only if no data from server is required)
- Most of the rendering happens on the client side
- Communication with the server in the background (very difficult with pure REST – multiple requests with hard to predict results)
- Confusing for users: Back/Refresh buttons cause disaster (lost work)
- Client architecture becomes important – a lot of code on the client
- Applying design patterns in your code gets even harder (they are baked into the frameworks)

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 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

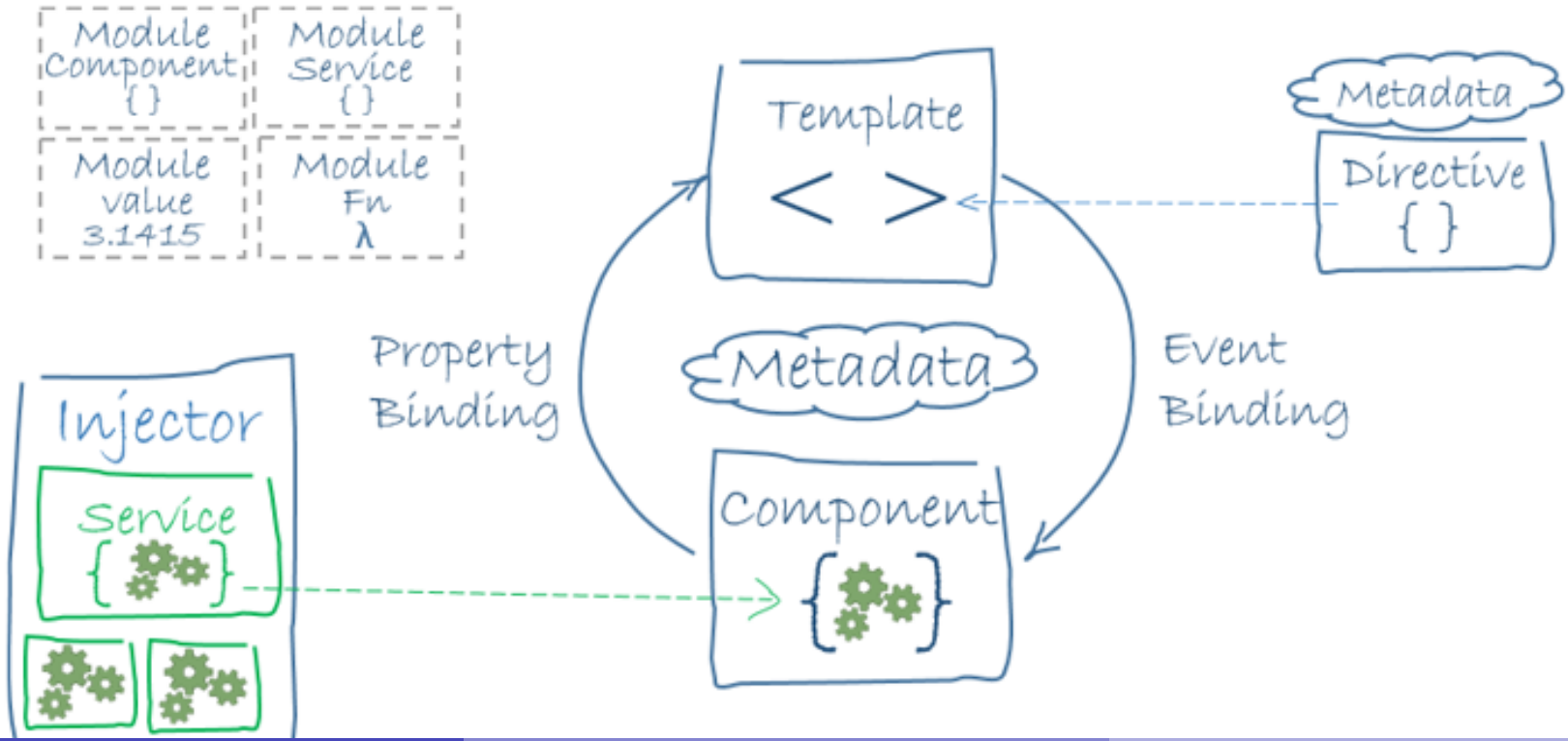
- Navigation and *Back* support
- Scroll history position
- Event cancelling (navigation)
- Bookmarking
- SEO (search engines see whole site as one page)[1]
- Whole app (many files) must be loaded before start[2]

[1] Unless using server-side rendering, which requires significant extra configuration.
[2] Unless using advanced techniques such as code splitting and lazy-loading.

Frameworks

Angular (2+)

- Developed by Google (but open-source)
- Completely rewritten since AngularJS (1.X), currently v15
- 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 at ~~Facebook~~ Meta (but open-source)
- Used extensively by Facebook and Instagram, but also Netflix, Uber, Microsoft (e.g. Teams)...
- High performance thanks to virtual DOM
- Leaves a lot to other libraries (routing, complex state, AJAX)
- XML-like JS syntax: JSX $\rightarrow$ transpilation (almost) inevitable
- React Native for developing native applications for iOS, Android and UWP in JS
- Easy to integrate into legacy web applications
- `https://facebook.github.io/react/`

React Example[3]

```
export default function MyComponent(props: MyProps) {
   const router = useRouter();
   const [data, setData] = useState<DataType>();

   useEffect(() => {
      restCall(props.id, response => setData(response.data), error => {
            })
   }, [props.idExamDate])

  return data.isVisible && (<div>{data}<div>);
}
```

[3] This example uses *Hooks*, which *let you use state and other React features without writing a class*. Available from React 16.8 onwards.

WebComponents

- Components (like in React)
- HTML + DOM standardized
- Supported by all browsers, no library needed
- Remains on client, no server-side support needed
- Plain JavaScript
- Simple code in TypeScript via annotations
- Used in GitHub.com `https://github.githubassets.com/assets/app/assets/modules/github/behaviors/batch-deferred-content.ts`
- `https://www.webcomponents.org/`, `https://developer.mozilla.org/en-US/docs/Web/Web_Components`, `https://github.com/aubi/sample-js-webelements`

WebComponents Example

```
import {LitElement, html, css, customElement, property} from 'https://
    unpkg.com/lit-element/lit-element.js?module';

class NamedayElement extends LitElement {
   constructor() {
      super();
      this.nameDay = 'loading...';
   }

   static get properties() {
      return {
         date: {type: String },
         dateDesc: {type: String },
         nameDay: {type: String}
      };
   }

   static get styles() {
      return css`.emph { color: green; }`;
   }
```

WebComponents Example (cont.)

```
    connectedCallback() {
        super.connectedCallback();
        this.getModel().then(res => {
            this.nameDay = res[0].name;
        });
    }

    async getModel() {
        var url = "https://svatky.adresa.info/json";
        this.dateDesc = "Today";
        var response = await fetch(url);
        return response.json();
    }

    render() {
        return html`${this.dateDesc} is the nameday for <span class="emph
            ">${this.nameDay}</span>`;
    }
}

customElements.define('nameday-element', NamedayElement);
in html:
<nameday-element dateDesc="Dnes" />
```

Other JS-based Alternatives

Vue

- *Approachable, performant and versatile* open source framework
- Similar to React in scope, performance and usage
- More template-oriented (not everything is JS) → better comprehensibility for designers, HTML developers
- Used at Adobe, Trivago, GitLab...
- `https://vuejs.org/`

```
<template>
  <p>{{ greeting }} World!</p>
</template>

<script>
module.exports = {
  data: function () {
    return {
      greeting: 'Hello'
    }
  }
}
</script>

<style scoped>
p {
  font-size: 2em;
  text-align: center;
}
</style>
```

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

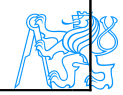
Frontend – Backend Communication

- 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));
            });
    };
}
```

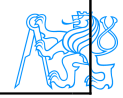
```
GET /eshop/rest/categories HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Accept: application/json
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
     /61.0.3163.91 Safari/537.36
```



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));
            });
    };
}
```

GraphQL

Figure: GraphQL

- "Not everything has to be RESTful..."
- Query languange instead of agreeing on API.
- Introduces security vulnerability, needs careful checking!

GraphQL vs REST

REST is quite verbose – many API calls may be required.
Let's load a person's profile[4]:

- `GET /user/144`
- `GET /user/144/friends`
- `GET /user/144/posts?limit=2`
- `GET /post/667/comments`
- `GET /post/1658/comments`

For dashboards with different services, it gets even worse...

The same query with GraphQL

POST /graphql

```
{
  profileById(id: 144) {
    name
    friends {
      id
      name
    }
    posts(limit: 2) {
      id
      content
      comments {
        id, content
      }
    }
  }
}
```

[1]This example is available at `https://gitlab.fel.cvut.cz/ear/graphql-demo`.

WSO2

- API Management (versioning, testing, thoughput, customization)
- Integration between many apps (independent vendors)
- Customer Identity (central)
- Access Management (central)

Client Architecture

Client Architecture

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

- 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.
- Used extensively in Android apps.

Conclusion

UI – Bits from Recent History

- REST + React.js are the most popular
- Flux/Redux are frequently used, useful for complex pages
- Rise of standard Web Components, vanilla JS/DOM 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)
- The rest is very unstable:
  - Angular is easy → Angular is hell, React is easy → React is hell, Vue is easy → Back to React → ???
  - 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 without extensive configuration.
- 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 developed). It was great for fat clients for Java team.
- **JSF** – Java programmers learn it quickly, easy to provide rich functionality, PrimeFaces actively developed, 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

Next.js

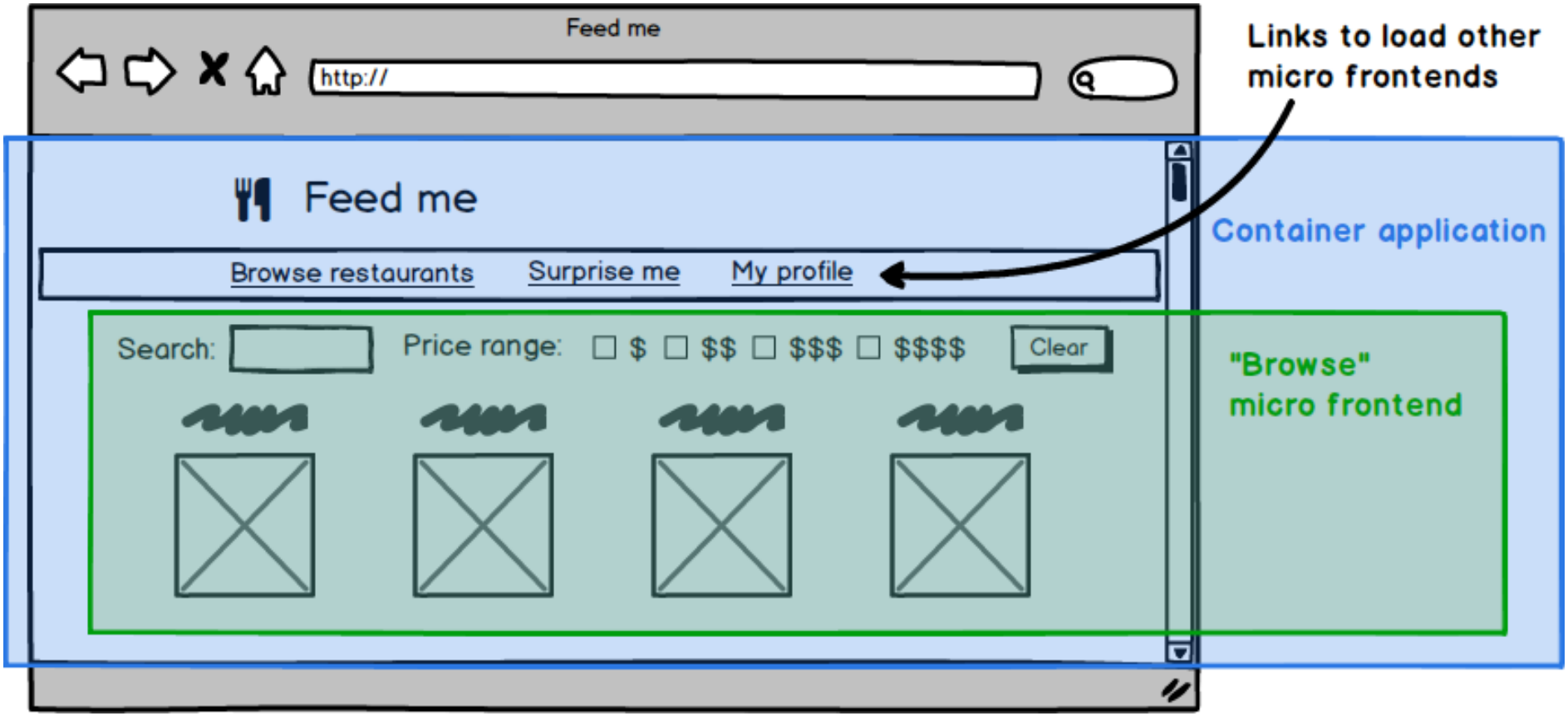
- Bookmarking, linking, SEO – large issues of many SPAs
- *What if we could generate the HTML on the server instead?*
- SSR[5]: (pre-)rendering React server-side **at request time** with `getServerSideProps`, then *hydrating* on the client
- SSG[6]: (pre-)rendering at **build** time (data must be available) → just HTML and JSON[7] → cacheable, very fast (e.g. static websites)
- Incremental Static Regeneration: "periodic SSG" (e.g. blogs)
- Also includes routing, API routes, logging, error handling...
- `https://nextjs.org/`

[5] Server-Side Rendering.
[6] Static Site Generation.
[7] Not entirely true, still gets hydrated client-side.

Micro Frontends – Introduction

- Applying the microservice architecture on FE, i.e. services are
  - highly maintainable,
  - organized around business capabilities,
  - loosely coupled and mostly self-contained,
  - independently deployable.
- Necessary, because large frontends
  - may consist of “modules” delivered by multiple teams (even vendors), each having a different technology stack and release schedule,
  - become notoriously hard to maintain,
  - should not be blindly split and thus feature duplicated common code,
  - ...

Micro Frontends – Splitting



Micro Frontends – Implementation

- We use Webpack 5 Module Federation[8] with React
- Each ES module may expose (parts of) itself and consume *remotes*, e.g. `http://localhost:3001/remoteEntry.js`
- Single container application (**shell**) provides user information and shared dependencies, such as framework and design system code
- Asynchronous lazy loading: `React.lazy(()=> import("nav/Header"))`
- Loader components spread throughout pages
- → `<Suspense />`
- Every *application module* is independently built and deployed

[8] `https://webpack.js.org/concepts/module-federation/`

Micro Frontend – Remarks

- MFEs are relatively fresh (2019, 2020+)
- Slow corporate adoption, huge potential – multi-vendor delivery, vertical teams with clear ownership (and SLAs)
- Ideal when coupled with microservice backends
- Yet another "magical" client-side concept

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,
- https://martinfowler.com/articles/micro-frontends.html.