# Web Application Clients

Petr Aubrecht (Martin Ledvinka, Miroslav Holeček)

petr@aubrecht.net

Winter Term 2025

# Contents

# REST – Management

# WSO2

- Situation: big complex system, SOA/microservices
- Multiple applications from independent vendors
- API Management (versioning, testing, thoughput control, customization)
- Integration between incompatible apps (WS/REST), adapters
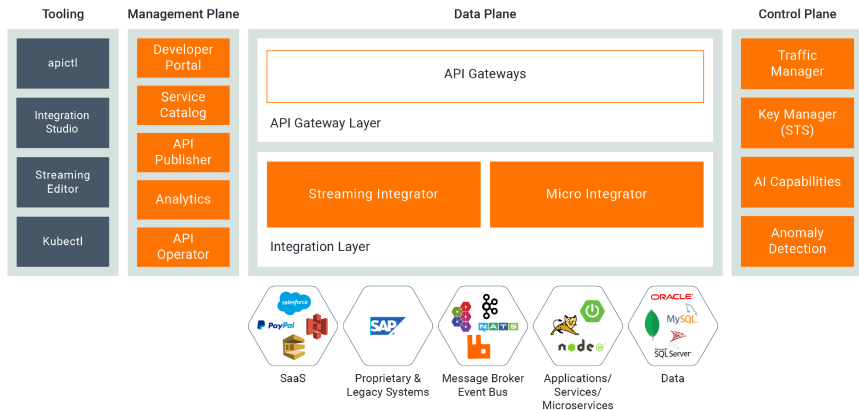- Access Management (central)

# WSO2 – Architecture



Figure: From https://apim.docs.wso2.com/en/4.0.0/get-started/apim-architecture/

# WSO2 – Gateway

- How data is transferred, converted, modified



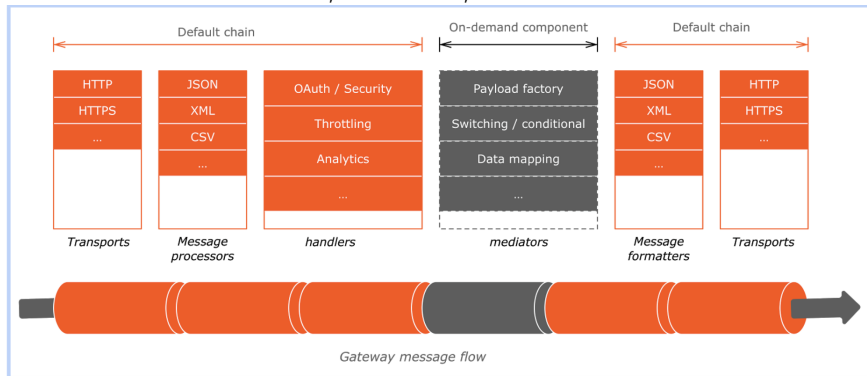Figure: From https://apim.docs.wso2.com/en/4.0.0/
get-started/apim-architecture/

# User Interface

## Petr Aubrecht's View

- UI is ALWAYS difficult (especially for Java developers), mostly hated...
- Who is expert in these technologies:

# Petr Aubrecht's View

- UI is ALWAYS difficult (especially for Java developers), mostly hated...
- Who is expert in these technologies:
    - (X)HTML

## Petr Aubrecht's View

- UI is ALWAYS difficult (especially for Java developers), mostly hated...
- Who is expert in these technologies:
    - (X)HTML
    - CSS

# Petr Aubrecht's View

- UI is ALWAYS difficult (especially for Java developers), mostly hated...
- Who is expert in these technologies:
    - (X)HTML
    - CSS
    - JS

# Petr Aubrecht's View

- UI is ALWAYS difficult (especially for Java developers), mostly hated...
- Who is expert in these technologies:
  - (X)HTML
  - CSS
  - JS
  - XSL-T + XSL-FO (PDFs)

## Petr Aubrecht's View

- UI is ALWAYS difficult (especially for Java developers), mostly hated...
- Who is expert in these technologies:
  - (X)HTML
  - CSS
  - JS
  - XSL-T + XSL-FO (PDFs)
- Even FB is often broken (image cannot be closed sometimes, see broken-facebook-image.png)
- **...but it is the best place for innovation!**

# Historical Overview

# Evolution of the Web

- Increasing complexity
- Plus Mozilla since 1998,[1] Netscape released source codes



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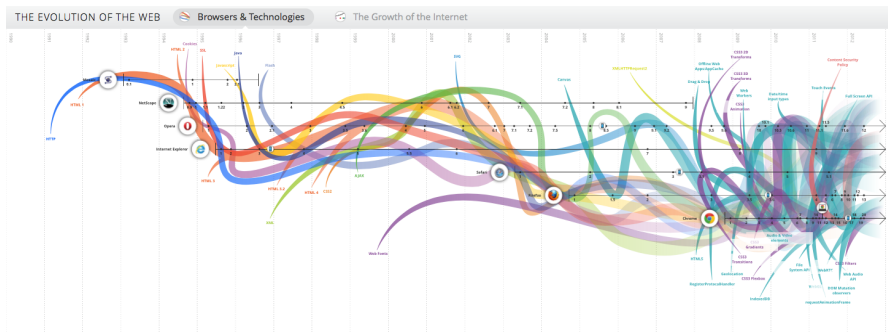
---

[1] https://www.mozilla.org/en-US/about/history/

# Web Applications

- A lot of options: CSS3, Table/CSS/Flexbox/Grid, libraries, approaches (classes, functions, hooks)...
- Browsers unified: FF, Chrome, Safari/WebKit
- 2025: HTML 5, only JavaScript
- WebAssembly is promising, but still not used to develop web apps except video calls or games

# Common Gateway Interface (CGI)

- Mid-1990s
- Dynamic content – server (httpd, Apache) starts a program
  - passes parameters in environment variables
  - stdout is returned to server and client
- It starts a program for every request
- 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)

```java
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
<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>
```
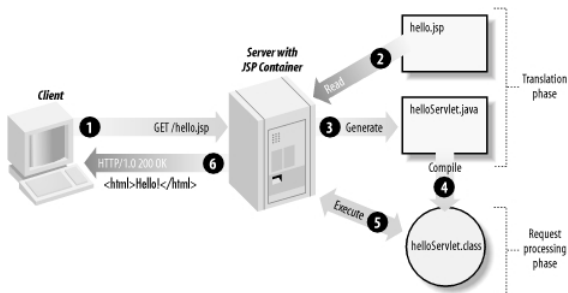
## Java Server Faces

- 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 – Plain HTML 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 |

# JSF Example IV – PrimeFaces Interactive Output

## 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

  Wicket Pages represented by Java class instances on server

  Active:

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

Spring MVC + Templates Servlet-based API with various UI technologies,
Thymeleaf, **FreeMarker**, Groovy Markup

# JavaScript-based UI

# JS-based UI Principles

- All rendering happens on the client side
- 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 them 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 months (jQuery, transpilation, Redux, classes vs hooks, Angular 2.0, micro front-ends, web components etc.)
- Incompatible implementations in browsers – 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.

```html
<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

- Motivation: instant changes of UI
- Reality: True if done carefully. Messy, shaking UI otherwise (Jira)
- View changes done by modifications of the DOM tree
- Use router – URL parameters (bookmarkable!) and internal state define content
- Communication with the server in the background (very difficult with pure REST – multiple asynchronous requests)
- Back/Refresh buttons cause disaster (lost work)

# 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.min.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 |

# Frameworks

# Angular (2+)

- Developed by Google (open-source)
- Completely rewritten since AngularJS (1.X, currently v21)
- Encourages use of MVC with two-way binding
- HTML templates enhanced with hooks for the JS controllers
- Built-in routing, AJAX
- https://angular.io/
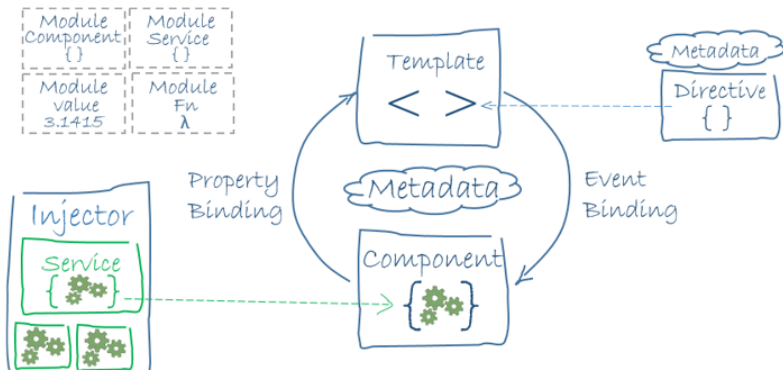
# Angular Example

```
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 (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 → transpilation (almost) inevitable
- React Native for developing native applications for iOS, Android and UWP in JS
- Easy to integrate into legacy web applications
- https://react.dev/

# React Example

- Using functional *Hooks* (React 16.8+), vs classes
- Includes TypeScript types

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

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

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

# Next.js with Page Router ($< 2023$)

- Builds on top of React
- Bookmarking, linking, SEO – large issues of many SPAs
- *What if we could generate the HTML on the server instead?*
- SSR:[2] (pre-)rendering React server-side **at request time**
- SSG:[3] (pre-)rendering at **build** time (data must be available) $\rightarrow$ just HTML and JSON[4] $\rightarrow$ cacheable, very fast (i.e. static websites)
- Incremental Static Regeneration: "periodic SSG" (e.g. blogs)
- Also includes routing, API routes, logging, error handling...
- https://nextjs.org/

---

[2]Server-Side Rendering.
[3]Static Site Generation.
[4]Not entirely true, still gets hydrated client-side.

# Next.js with App Router

- *React Server Components*
- Further blurring the line between SPAs and... not
- Newer terminology, similar patterns
- `"use client"`; `"use server"`; directives in modules, functions
- CVE-2025-55182 (`https://react2shell.com/`)

# 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
- 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/

```html
<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/

```javascript
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
- Usually handled by a library nowadays (SWR, TanStack Query)

# Frontend – Backend Communication Example

```javascript
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 vulnerabilities, needs very careful design and checking!

# GraphQL vs REST

REST is quite verbose – many API calls may be required.
Let's load a person's profile[5]:

- `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

```
1 ▾ {
2 ▾   profileById(id: 144) {
3       name
4 ▾     friends {
5         id
6         name
7       }
8 ▾     posts(limit: 2) {
9         id
10        content
11 ▾      comments {
12          id, content
13        }
14      }
15    }
16  }
```

---

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

# 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.

# 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
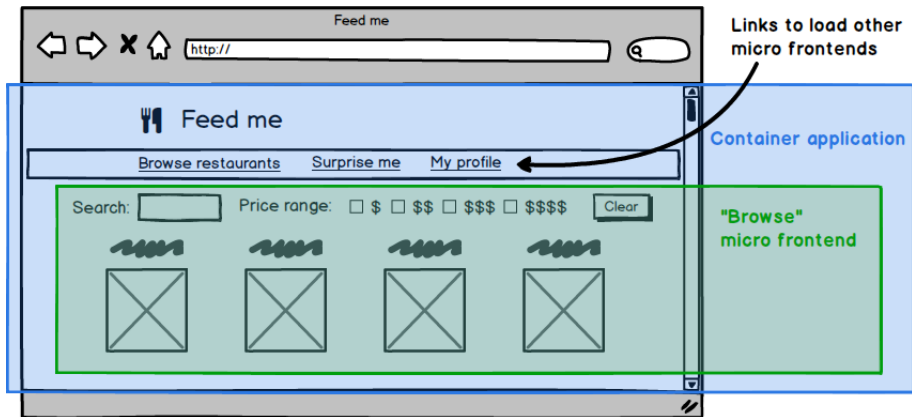
# Micro Frontends – Implementation

- We use Webpack 5 Module Federation[6] with React and Rsbuild
- 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
- $\rightarrow$ `<Suspense />`
- Every *application module* is independently built and deployed

---

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

# Micro Frontends – Remarks

- MFEs are relatively fresh (2019 MF v1, 2024 MF v2)
- Slow corporate adoption, huge potential – multi-vendor delivery, vertical teams with clear ownership (and SLAs)
- Ideal when coupled with microservice backends
- Yet another "magical", hard-to-debug client-side concept
- Facebook, GitHub, TikTok, . . .

# Conclusion

# P.A.'s Experience

- REST + React.js are the most popular
- JS + REST needs two backend and frontend teams, which quickly separate and hate each other
- JS is most frequently used language. And most hated ever, Java developers prefer only backend.
- JS requires every data exchange visible via REST, much more work and vectors for attack
- In JS, revolution happens every half a year, no stable best practices (opinions change frequently) – what is the lenght of your application's life – 1 year, 5 years, 10 years?
- Fullstack – GWT, JSF or JS + node.js
- Appropriate for sites with millions of users like Facebook.

# Summary

- **GWT** – perfect for Java programmers, full type-check, all Java, deprecated (Vaadin is still alive). 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, WebSockets). Great for typical information systems.
- **JS, React.js** – basics are simple, complexity rapidly grows
- Alternatives
    - **Web Components** + vanilla JS
    - **WebAssembly** – possibility to run code other than JS, possibly Java

# 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.