WEB DESIGN AND PROGRAMMING

DECOUPLED WEB ARCHITECTURE
6

FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

ASST. PROF. DR. SAMET KAYA

skaya@fsm.edu.tr

kysamet@gmail.com

Why Decouple Our Architecture?

- In our previous approach, we built monolithic applications using Spring Boot and Thymeleaf.
- In this section, we will separate the frontend and backend to run as two distinct applications.
- Our focus: Setting up the frontend development environment with React.

React and Node.js

- **React**: A component-based User Interface (UI) library developed by Facebook. It runs in the browser (client-side).
- **Node.js**: Although React runs in the browser, we need Node.js as the engine for our development environment. It is not the server our application runs on.

The Role of Node.js in Development

- **Package Management (npm):** Allows us to install external libraries like axios or react-router. npm is the direct equivalent of Maven/Gradle in the Java world.
- **Development Server**: Provides a local server (like localhost:3000) that allows us to see our code live in the browser as we write it.
- **Build Tools**: Runs tools (like Babel, Vite) that transpile our modern JavaScript (JSX, ES6+) into a single, optimized file that all browsers can understand.

Prerequisite: Installing Node.js

- Node.js must be installed before starting React development.
- Go to:
  - https://nodejs.org/
  - Recommended: Install **the LTS (Long Term Support)** version.
- **Verification (Terminal):**
  - node -v (e.g., v20.11.0)
  - npm -v (e.g., v10.2.4)

IntelliJ vs. VS Code

- **Option A: IntelliJ IDEA Ultimate (Paid):** Ideal for managing both Java (Spring Boot) and React projects in the same IDE with first-class support.
- **Option B: Visual Studio Code (Free):** The industry standard for frontend developers. This is the smartest choice if you use IntelliJ Community Edition (as its modern JavaScript support is limited).
- **Note**: We will run commands via the terminal, so either IDE is fine.

From Server-Side to Client-Side Rendering

- **Traditional Approach (Thymeleaf):**
  - Server-Side Rendering (SSR): The server (Spring Boot) combines data with HTML templates.
  - Result: The browser receives a fully painted HTML page.
  - Drawback: Every page navigation requires a full reload from the server.
- **Modern Approach (React):**
  - **Client-Side Rendering (CSR):** The server sends an empty HTML skeleton and a JavaScript bundle.
  - **Result**: The browser (Client) builds the UI dynamically.
  - **Benefit**: Faster interactions, feels like a desktop app (Single Page Application).

How React Works as a View Engine

- **The Single Entry Point:**
  - Unlike Thymeleaf, we don't have multiple .html files for every page.
  - We have one single index.html file with an empty container: .
- **The Injection Logic:**
  - React takes control of this specific div.
  - It injects the entire application component tree into this root element.
- **Code Perspective (main.jsx):**
  - ReactDOM.createRoot(document.getElementById('root')).render(<App />)
  - This line tells React: "Find the element with id 'root' and display my App inside it."

Under the Hood: The Virtual DOM

- **The Problem with Real DOM(Document Object Key-Value):**
  - Updating the actual HTML directly (like jQuery or traditional JS) is slow and performance-heavy.
- **The React Solution (Virtual DOM):**
  - React keeps a lightweight copy of the DOM in memory.
  - **Diffing Algorithm**: When data changes, React compares the Virtual DOM with the previous version.
  - **Reconciliation**: It updates **only** the changed parts in the real browser DOM, not the entire page.
- **Efficiency**: This minimizes screen repainting and maximizes performance.

JSX - JavaScript XML

- **JSX (JavaScript XML):** A syntax extension for JavaScript that looks like HTML but is actually JavaScript.
- Developed by Facebook as part of React.JSX makes it easier to write and visualize the UI structure.
- Important: JSX is NOT HTML. It's syntactic sugar for React.createElement() calls.
- Key Point: Browsers don't understand JSX. Tools like Babel transpile JSX into regular JavaScript.

```
// JSX Syntax
const element = <h1>Hello, World!</h1>;

// Equivalent JavaScript (what JSX compiles to)
const element = React.createElement('h1', null, 'Hello, World!');
```

JSX Rules and Syntax - 1

- **Single Parent Element**: JSX must return a single root element.

```
// ❌ Wrong
return (
  <h1>Title</h1>
  <p>Paragraph</p>
);

// ✅ Correct
return (
  <div>
    <h1>Title</h1>
    <p>Paragraph</p>
  </div>
);
```

JSX Rules and Syntax - 2

- **JavaScript Expressions:** Use curly braces {} to embed JavaScript.

```
const name = "John";
const element = <h1>Hello, {name}!</h1>;
```

- Attribute Naming: Use camelCase (className instead of class, onClick instead of onclick).

```
<div className="container" onClick={handleClick}>
```

- Self-Closing Tags: All tags must be closed.

```
<img src="logo.png" />
<input type="text" />
```

First React Project (with Vite)

- We are using Vite (pronounced "veet") for its speed, replacing the older create-react-app (CRA).
- **Command:**
  - npm create vite@latest
- **Interactive Prompts:**
  - Project name: react-frontend
  - Select a framework: React
  - Select a variant: JavaScript

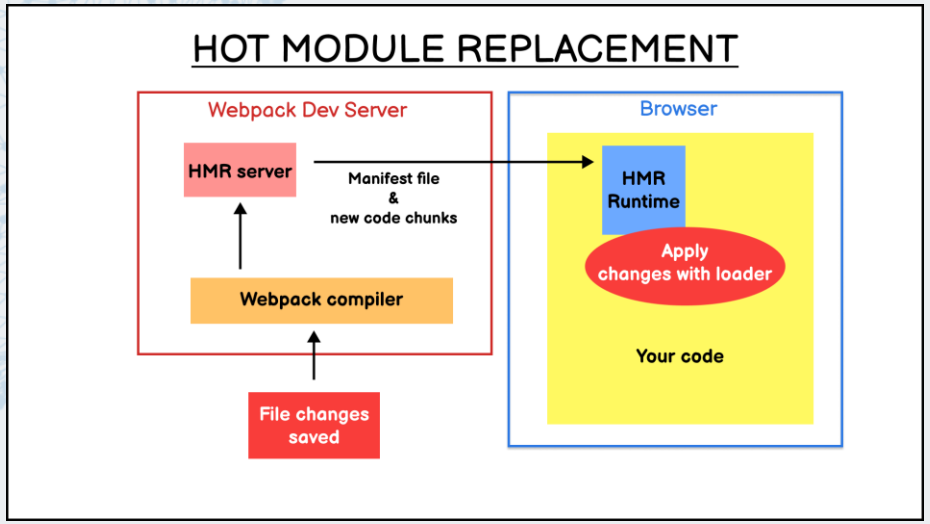
Organizing the Project Structure Content

- Scalability: As the application grows, keeping everything in src becomes unmanageable. We use a "Feature" or "Type" based structure.
- **src/components:** Reusable, stateless UI elements (e.g., Navbar, Footer, Button).
- **src/pages:** Components that represent full views/routes (e.g., HomePage, ProductPage).
- **src/services:** Centralized location for API calls (Axios configuration).
- **src/assets:** Static files like images and global CSS.

```
src/
 ├── components/  (Navbar.jsx, ProductCard.jsx)
 ├── pages/       (Home.jsx, Products.jsx)
 ├── services/    (api.js)
 └── App.jsx
```

HMR (Hot Module Replacement)

- **HMR (Hot Module Replacement):** One of the most powerful features of React and Vite.
- Open src/App.jsx and change the content (e.g., <h1>Hello World</h1>).
- The moment you **Save (Ctrl+S)** the file, the page in the browser updates instantly without a full page refresh.
- This feature dramatically speeds up development.



React API Client: Why Axios

- While React has a built-in fetch function, the industry standard is the Axios library.
- Axios is a popular, promise-based HTTP client.
- **Key Features:**
  - Promise-based (supports async/await).
  - Works in both the browser and Node.js.
  - Automatically transforms incoming JSON into JavaScript objects.
  - **Interceptors**: Its most powerful feature.
  - Lets you "intercept" requests (e.g., to add a token) or responses (e.g., for global error handling).

Axios Installation and Basic Usage

- Installation (In React Project Terminal):
  - npm install axios
- GET Request (with Async/Await):

```
import axios from 'axios';

async function getUser() {
    try {
      const response = await axios.get('/user?ID=12345');
      console.log(response.data);
    }
    catch (error) {
      console.error(error);
    }
}
```

- POST Request:

```
axios.post('/user', { firstName: 'Fred', lastName: 'Flintstone' })
  .then(response => {
    console.log(response.data);
  });
```

The Heart of React: Props vs. State

1. **Props (Properties):**
   - How data is passed from a parent component to a child component.
   - **Rule**: They are Read-Only. A child component must never modify the props it receives (One-Way Data Flow).
   - **Analogy**: Parameters passed to a Java method.
2. **State:**
   - A component's private, internal memory that can change over time. Defined with the **useState hook**.
   - **Rule**: When state changes, React automatically re-renders that component.
   - **Analogy**: Private fields in a Java class.

Stateful vs. Stateless Components

- Stateful / Container Components:
  - The "smart" components.
  - They fetch data from APIs (useEffect), hold data in their state (useState), and execute business logic.
- Stateless / Presentational Components:
  - The "dumb" components.
  - They have no state of their own (or very little).
  - Their only job is to receive data via props and display it nicely. They don't know where the data came from.
- Benefit: This architecture makes code reusable, testable, and easier to maintain.

The Data Fetching Pattern


```
function ProductList() {
  // 1. State to store data
  const [products, setProducts] = useState([]);

  // 2. Effect to trigger fetch on load
  useEffect(() => {
    axios.get("http://localhost:8080/api/products")
      .then(response => {
        // 3. Update state with data from Backend
        setProducts(response.data);
      })
      .catch(error => console.error(error));
  }, []); // Empty array ensures this runs only once

  return (
    <ul>
      {products.map(p => <li key={p.id}>{p.name}</li>)}
    </ul>
  );
}
```

- To fetch data from our Spring Boot API, we follow a standard 3-step pattern:

1. **useState**: Initialize a state variable to hold the incoming data (usually an empty array).
2. **useEffect**: Trigger the API call immediately when the component mounts (loads).
3. **axios**: Perform the asynchronous HTTP GET request.

Client-Side Routing

- **Instillation:**
  - npm install react-router-dom
- **SPA Logic:** In a Single Page Application, we do not reload the page to change views. We swap components.
- **react-router-dom:** The standard library for routing in React.
- **Key Components:**
  - **<BrowserRouter>:** Wraps the app to enable routing.
  - **<Routes>:** A container for all route definitions.
  - **<Route>:** Maps a URL path (e.g., /products) to a specific Component.

Router


```jsx
// App.jsx
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import HomePage from './pages/HomePage';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/products" element={<ProductPage />} />
      </Routes>
    </BrowserRouter>
  );
}
```

Bridging the Architectures

- **Two Worlds, Two Ports**
- We now have two servers:
  - **Backend (Spring Boot)**: localhost:8080 (The JSON API server)
  - **Frontend (React):** localhost:5173 (The Vite development server)
- This is a **Decoupled Architecture**.
- The React app at localhost:5173 must make an API call to localhost:8080.

The Inevitable Problem: CORS Error

- **Cross-origin resource sharing -> CORS**
- When React tries to fetch data from the API, the browser (F12 Console) will show this error:
  - Access to XMLHttpRequest at 'http://localhost:8080/...' from origin 'http://localhost:3000' has been blocked by CORS policy...
- Why? Browsers have a security rule called the Same-Origin Policy (SOP).
- For security reasons, the browser blocks requests coming from a different origin (in our case, a different port).

How CORS Works

- For "non-simple" requests like PUT or DELETE, the browser sends a "Preflight" request before sending the actual request.

1. **Browser (Preflight):**
   - OPTIONS /api/data HTTP/1.1
   - Origin: http://localhost:5173
   - Access-Control-Request-Method: PUT
2. **Server (Response):**
   - Access-Control-Allow-Origin: http://localhost:5173
   - Access-Control-Allow-Methods: GET, PUT, POST, DELETE
   - If permission is granted, the browser then sends the actual PUT request.

The Solution: CORS (Cross-Origin Resource Sharing)

- **CORS**: A security mechanism that allows a browser to control access to requests from a different origin.
- **The Fix**: The server (Spring Boot) must tell the browser, "It's okay, I trust requests coming from localhost:5173.
- "We must add this permission to our **Spring Boot project**.

Spring Boot Global CORS Configuration

- In our Spring Boot project, add the following @Bean to a SecurityConfig file:
  - @Bean CorsConfigurationSource corsConfigurationSource() { ... }
- **Configuration Details:**
  - `CorsConfiguration config = new CorsConfiguration();`
  - `config.setAllowedOrigins(List.of("http://localhost:5173"));`
  - `config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));`
  - `config.setAllowedHeaders(List.of("*"));`
  - `config.setAllowCredentials(true);`
  - `source.registerCorsConfiguration("/api/**", config);`
- Activate it by calling .cors() in your SecurityFilterChain.

Alternative: The @CrossOrigin Annotation

- Instead of global configuration, you can grant permission on individual Controllers or methods.

```
@CrossOrigin(origins = "http://localhost:5173", allowCredentials = "true")

@RestController

@RequestMapping("/api")

public class MyController { ... }
```

- Restart Spring Boot, and the CORS error will be resolved.

@CrossOrigin Annotation Rest


```
@RestController
@RequestMapping("/api/products")
// Allow requests from the React Frontend
@CrossOrigin(origins = "http://localhost:5173")
public class ProductController {

    @Autowired
    private ProductService productService;

    @GetMapping
    public List<Product> getAllProducts() {
        return productService.findAll();
    }

    @PostMapping
    public Product createProduct(@RequestBody Product product) {
        return productService.save(product);
    }
}
```

Product Management App

- Create React app with Vite

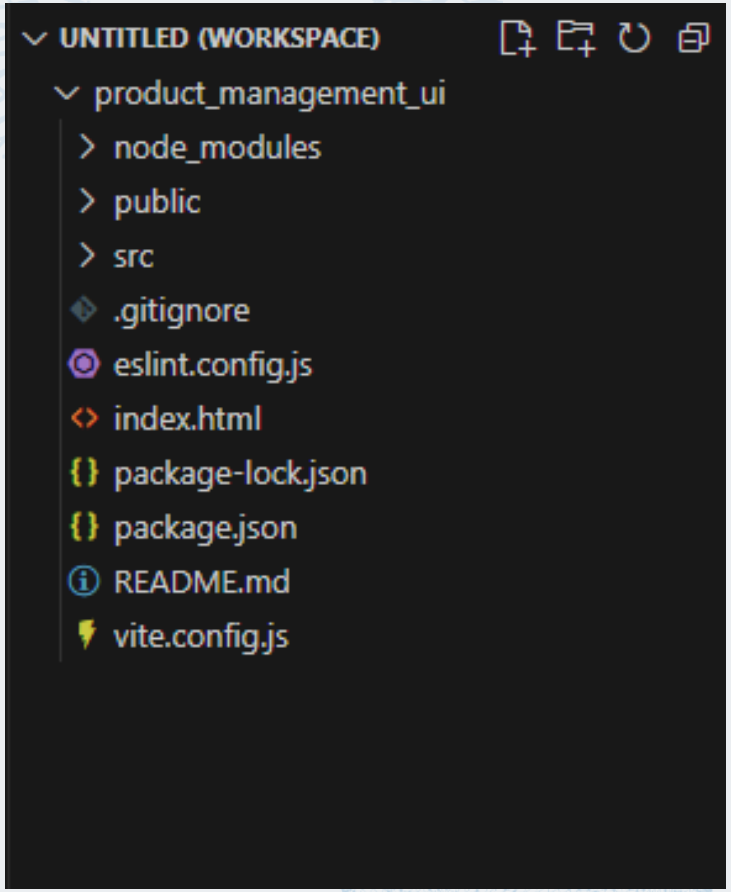
```
npx create-vite@latest product_management_ui --template react
```

```
cd product_management_ui
```

```
npm install axios react-router-dom
```

- Terminal:

```
→ Local:   http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```



ASST. PROF. DR. SAMET KAYA

FATIH SULTAN MEHMET VAKIF UNIVERSITY

skaya@fsm.edu.tr

kysamet@gmail.com

Some images and icons in this presentation are sourced from Freepik.