Lecture 18
Building Large Web Apps

Web Frameworks, React Native, and the Future of Web Programming

05-431/631 Software Structures for User Interfaces (SSUI)

Fall, 2022

Overview

- We have primarily looked at **libraries** for building interactive UIs such as **React**
- Larger, complex **web applications** can benefit from frameworks that support
  - Routing
  - Server-side rendering
- Can we also move beyond websites to build **fast** and **mobile-native** UIs?

Today

- **Web Application Frameworks -** Next.js, SvelteKit
- **Native Frameworks -** React native
- Future of web development

Web Frameworks

Library vs. Framework

Library

- API of functions that help developers write applications

Framework

- Formal structure for guiding application development

Recap of History of Web Development

Historically, most websites were primarily hosted ***and generated*** by a backend

- **PHP** server that sends static websites for each request
  - /shirt/cmu -> new static website
- Backend templating service (**Django/Flask**) sending templated HTML files

Limitation: Limited interaction on the frontend! Can't reactively update state

Modern State of Web Development

Frameworks like **React** moved everything to the frontend

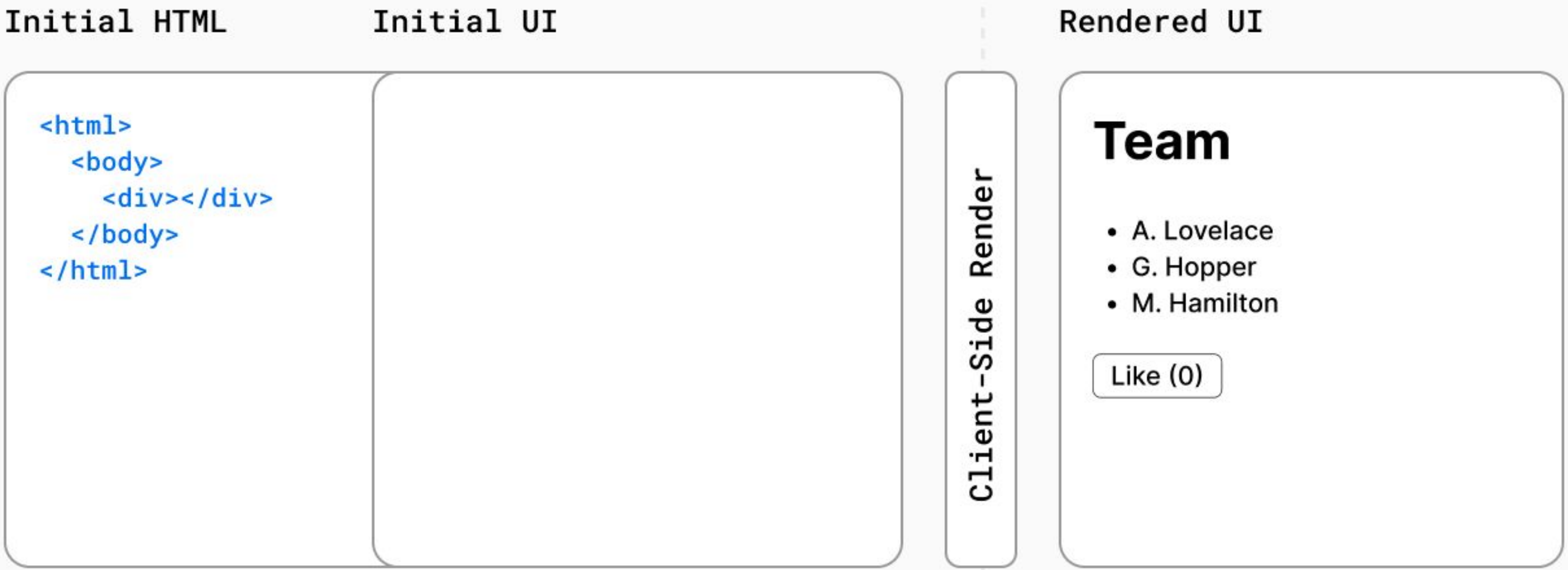
- Send **blank** HTML with JS to render DOM
- Can interact and manipulate state on frontend, only sync with backend
- Made fast, interactive applications possible

Aside: How ReactRouter works

With React, even routing is done in the frontend!

- When you navigate to `mysite.com/about` it doesn't get a new HTML file from the server
- ReactRouter *changes the URL you see in place* and renders new components, without navigating to a new HTML site
- Everything is from **one URL**, with Javascript updating the DOM

Client-Side Rendering



Initial HTML

```
<html>
  <body>
    <div></div>
  </body>
</html>
```

Initial UI

Client-Side Render

Rendered UI

Team

- A. Lovelace
- G. Hopper
- M. Hamilton

Like (0)

Limitations of Client-Side Rendering

Client-side rendering has its own downsides

- Can be slow to startup - have to download KBs of JS code to see anything
- Can be worse for indexing by Google (SEO) - Google indexes static websites more frequently than those generated by JS

Solution

- Combination of server-side rendering for fast loading and startup with client-side rendering for fast interactivity and reactivity

Example

- Look at source
- Disable JavaScript
- Throttle internet

Solution: Pre- and Server-side rendering

Modern frameworks **combine** client and server-side rendering

1. Pre-render HTML with initial state **on the server** (Pre-rendering)
2. Send the plain HTML file to the **client**
3. Send the **JS** files for interactivity to the **client**
4. **Hydrate** the plain HTML with interactive elements using the **JS**

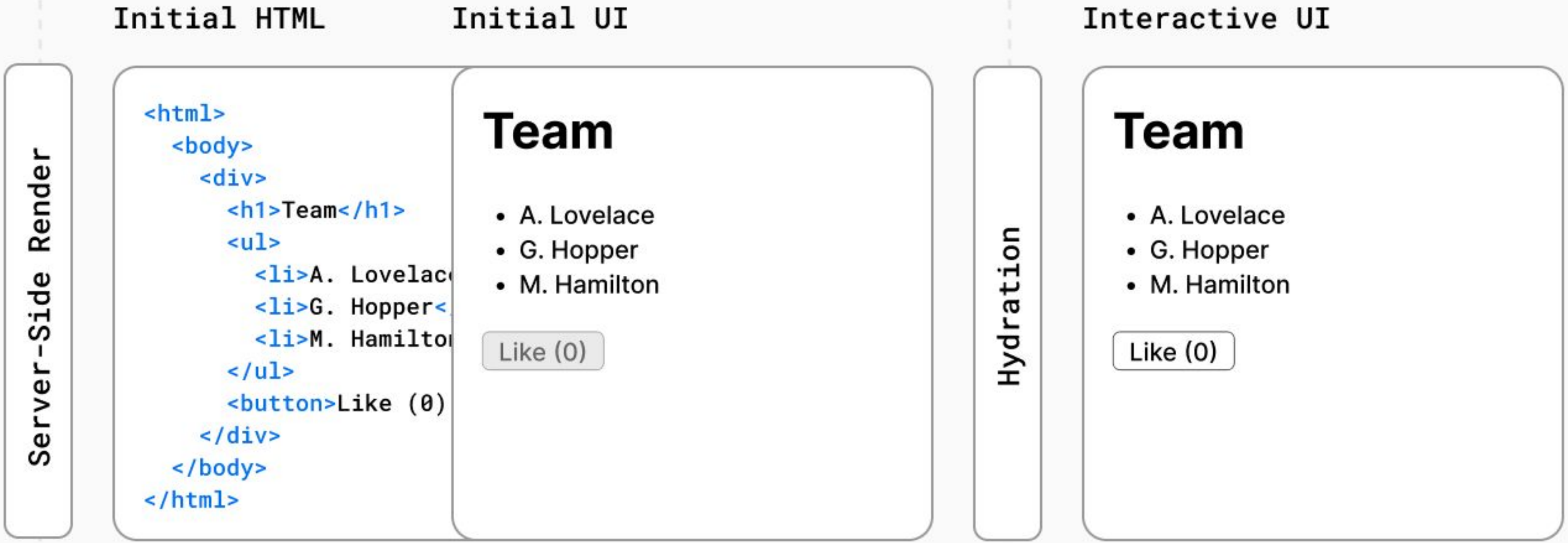
Hydration

With server-based rendering, when the JS gets sent to the server it has to:

1. Look at the HTML and figure out which elements have to be made reactive
2. Add listeners, update functions, and generate virtual DOM

This process is called **hydration -** filling the static HTML with interactive elements

Pre-Rendering



Initial HTML

Initial UI

Interactive UI

Server-Side Render

```
<html>
  <body>
    <div>
      <h1>Team</h1>
      <ul>
        <li>A. Lovelac
        <li>G. Hopper<
        <li>M. Hamilto
      </ul>
      <button>Like (0)
    </div>
  </body>
</html>
```

Team

- A. Lovelace
- G. Hopper
- M. Hamilton

Like (0)

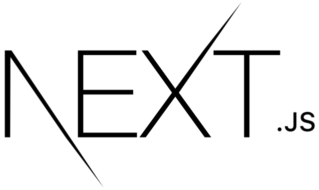
Hydration

Team

- A. Lovelace
- G. Hopper
- M. Hamilton

Like (0)

Next.JS



React-based framework for full web applications

- Pagination (routes)
- Server-side rendering
- Optimized serving

Used primarily for moving past single-page apps (SPAs) to larger applications

Next.js: Static vs. Server Side Rendering

Next provides two ways to pre-render components on the server

Static -> `getStaticProps()`
Render a page when **building** your app. The same for each user, e.g. an About page

Server-side rendering -> `getServerSideProps()`
Render a page on **every request.** Changes for each user, e.g. profile

Example

https://codesandbox.io/p/sandbox/jolly-shaw-v290m0

Other Frameworks: Nuxt, SvelteKit

There are similar web application frameworks for **Vue** and **Svelte**

- **Vue** has **Nuxt.js -** Similar file-based routing, server-side rendering
- **Svelte** has **SvelteKit -** Also file-based routing, server-side rendering

Native Mobile Apps

React Native

React Native

React Native is a library for writing smartphone applications using React

- JSX is a ***virtual*** representation of the DOM which is written to DOM elements
- ***Idea:*** Why not adapt react to write native components for different devices?
- Developers write React, which can be rendered to native IOS and Android code!

React Native

In order to work across platforms, does not support HTML tags in JSX

Only has a set of **native components** including

- Text
- Image
- List

Other components are available as NPM packages which implement native APIs

Example

What's Next?

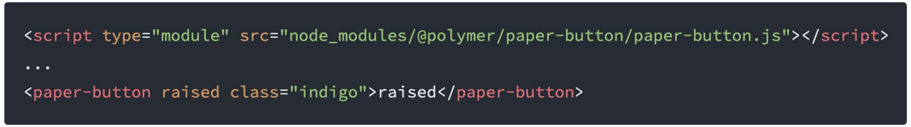
WEB COMPONENTS

Web Components: Native HTML Components

As we have seen, every modern UI library uses **components.**

Web Components is an attempt to make native, HTML-like components

- Use JS to imperatively define DIVs and behaviors
- Export as WebComponent
- Use natively in HTML



```
<script type="module" src="node_modules/@polymer/paper-button/paper-button.js"></script>
...
<paper-button raised class="indigo">raised</paper-button>
```

Some controversy on whether these are useful and implemented correctly:
https://dev.to/richharris/why-i-don-t-use-web-components-2cia

WebAssembly: Speeding up the Web

Libraries are enabling more complex web apps, but they are still based in JavaScript, a relatively slow and interpreted language

WebAssembly is a

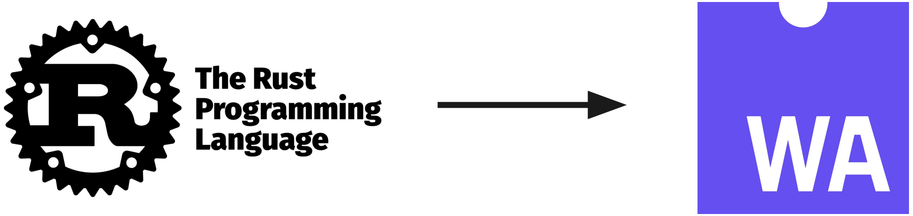
- Binary format for executable programs supported by web browsers
- And a library for interfacing with web APIs

All major browsers support it, and it enables high-performance applications by running directly on your CPU

WebAssembly

Code for WebAssembly can be written in many languages and is then **compiled down** into the binary format which can be called by the browser.

JavaScript then loads and invokes WebAssembly code



Examples of WebAssembly

- Figma is actually written in C++, and compiled and run using WebAssembly
- LiChess uses WebAssembly to run their chess engine
- In-memory databases like DuckDB can run in-memory in the browser

Final project idea: Speed up our editor using WASM, e.g. floodfill

WebGPU

A similar project is **WebGPU** for making GPU computation directly available in JS

It can be used for an array of purposes such as:

- Games
- Machine learning
- 3D graphics editing