



**OST**

Eastern Switzerland  
University of Applied Sciences

# Distributed Systems (DSy)

## Web Architecture

Thomas Bocek

20.03.2023

# Learning Goals

- Lecture 5
  - What are the options to build my challenge task?
  - What is currently “state-of-the-art”?

# Server-Side Rendering

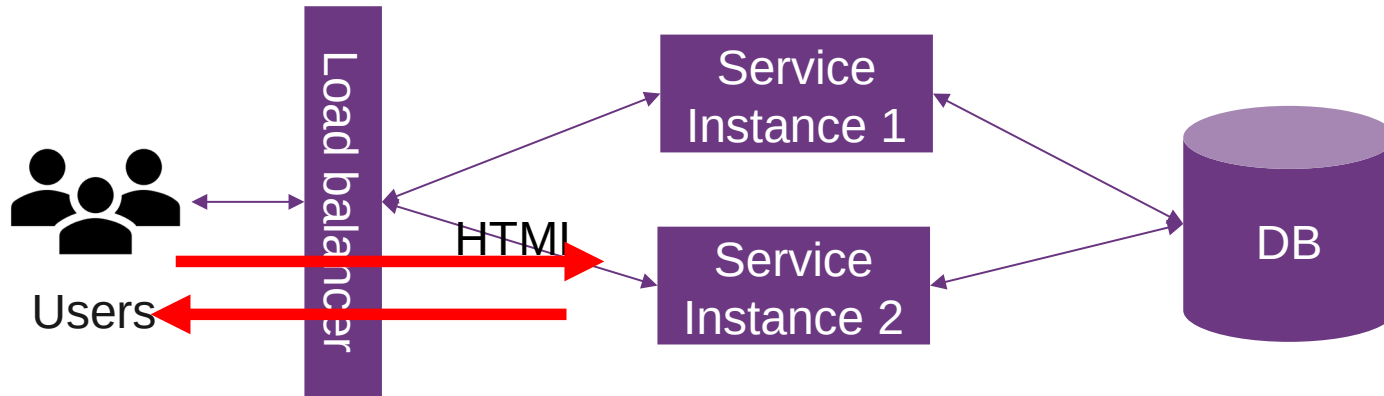
- “Classic” approach - “SSR”
- Not to be confused with static site rendering and (SSR)
- Server generates HTML/JS/CSS dynamically, sends the assets in real-time to the browser
  - User request: browser sends a request to the web server (server-side routing)
  - Server processing: server processes request by running server-side code (e.g., C#, Java, ...),
    - Fetch required data from a database or other sources
    - Server-side code can use template engines to render the HTML - reusability
  - Response: Generate the appropriate HTML, CSS, and JavaScript for the requested page.
  - Browser rendering: browser receives response and renders page
- Big advantage: SEO, but needs rendering for every request (caching!)
- Static site rendering: pre-render HTML/CSS/JS since its the same for every user. Done only once, resp, if the content changes.
  - [dsl.i.ost.ch](http://dsl.i.ost.ch) → markdown to HTML
  - Can also include DB access

# Single Page Application SPA / CSR

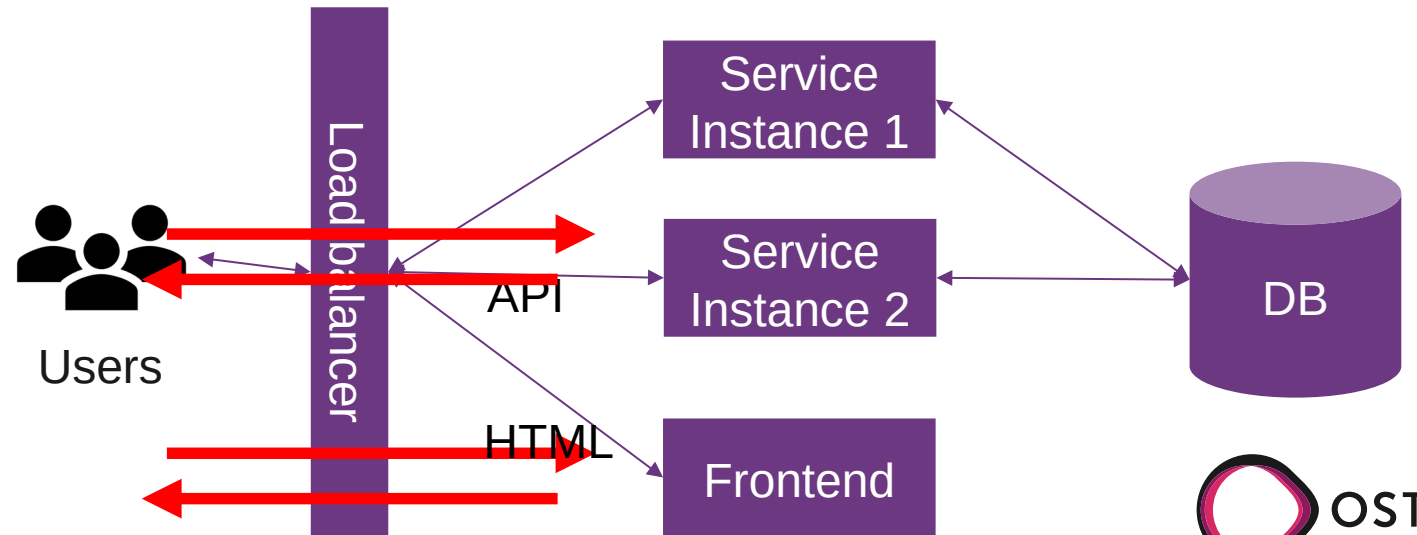
- Interactions occur within a single web page
- Client page dynamically updates as the user interacts with it, providing a smooth, app-like experience
- Relies on JavaScript to update UI
  - Initial request: browser sends a request to web server, hosting the HTML/JS
  - Initial response: server returns a single HTML file with CSS/JavaScript. JavaScript files contain the application's logic
  - Browser rendering: shows HTML file, typically a spinner, then executes JavaScript
- User interactions: JavaScript manages the UI updates. Application does not require full page reloads.
- API communication: When the SPA needs to fetch or send data, communicates through APIs
- Client-side routing: SPAs for navigation
- Use a framework: React, Angular, Vue
- Feels more app like
- The backend serves API requests only
- SEO only works if JavaScript is executed at the SE.

# Architecture

- Server side rendering (SSR)

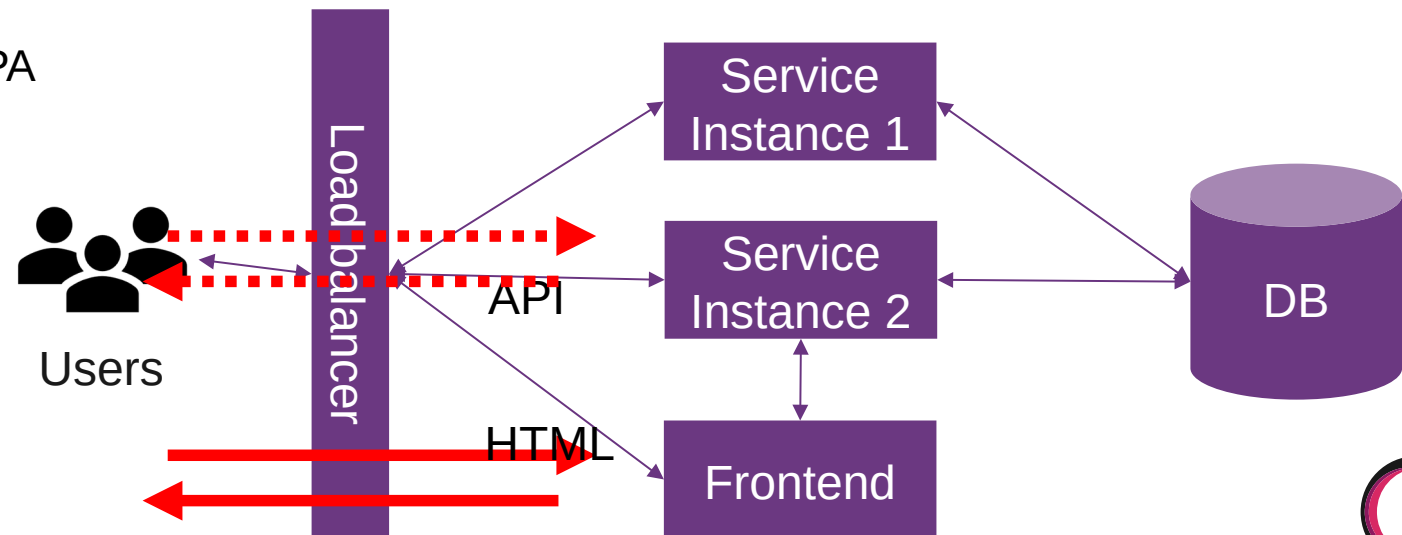


- Single page application (SPA), client side rendering (CSR)



# Web Architectures

- SPA: CORS - Cross-Origin Resource Sharing
  - HTTP-header based mechanism to indicate other origins (domain, scheme, or port) from which a browser can load assets.
- “State-of-the-art”: hydration
  - Initial HTML not with a “spinner”, but already the first content in HTML, like SSR (e.g., next.js server renders it for you - JavaScript)
  - Further access, with API, like SPA
  - Combine SSR/SPA
  - Flatfeestack: pre-SSR/SPA
    - Every user sees the same page, SSR can be pre-hydrated
- 17.03.2023: New React docs pretend SPAs don't exist anymore [\[link\]](#)
  - “The strongly recommended way to start a new React project is to use a framework such as Next.js, while the traditional route of using bundlers like Vite or CRA is fairly strongly discouraged.”





# Examples

- Static site rendering: [dsl.i.ost.ch](http://dsl.i.ost.ch)
  - Components: nginx
  - Java daemon who reacts on file changes in a director. If markdown file changes → create HTML, copy it to nginx directory
- Server side rendering (e.g., handlebarsjs)
  - Simple example: [ssr.go](http://ssr.go) (no template)
  - Components: go-based server
- SPA
  - Components: node server, go server

- Hydration
  - Best of both worlds, but adds complexity, needs JavaScript in the backend
  - E.g., react: `hydrate()` instead of `render()` method – choices... [source](#)

	Server Rendering	"Static SSR"	SSR with (Re)hydration	CSR with Prerendering	Full CSR
Overview:	An application where input is navigation requests and the output is HTML in response to them.	Built as a Single Page App, but all pages prerendered to static HTML as a build step, and the JS is <b>removed</b> .	Built as a Single Page App. The server prerenders pages, but the full app is also booted on the client.	A Single Page App, where the initial shell/skeleton is prerendered to static HTML at build time.	A Single Page App. All logic, rendering and booting is done on the client. HTML is essentially just script & style tags.
Authoring:	Entirely server-side (request, response, HTML)	Built as if client-side (components, DOM*, fetch)	Built as client-side	Client-side	Client-side
Rendering:	Dynamic HTML	Static HTML	Dynamic HTML and JS/DOM	Partial static HTML, then JS/DOM	Entirely JS/DOM
Server role:	Controls all aspects. (then client)	Delivers static HTML	Renders pages (navigation requests)	Delivers static HTML	Delivers static HTML
Pros:	<ul style="list-style-type: none"> <li>👉 TTI = FCP</li> <li>👉 Fully streaming</li> </ul>	<ul style="list-style-type: none"> <li>👉 Fast TTFB</li> <li>👉 TTI = FCP</li> <li>👉 Fully streaming</li> </ul>	<ul style="list-style-type: none"> <li>👉 Flexible</li> </ul>	<ul style="list-style-type: none"> <li>👉 Flexible</li> <li>👉 Fast TTFB</li> </ul>	<ul style="list-style-type: none"> <li>👉 Flexible</li> <li>👉 Fast TTFB</li> </ul>
Cons:	<ul style="list-style-type: none"> <li>👉 Slow TTFB</li> <li>👉 Inflexible</li> </ul>	<ul style="list-style-type: none"> <li>👉 Inflexible</li> <li>👉 Leads to hydration</li> </ul>	<ul style="list-style-type: none"> <li>👉 Slow TTFB</li> <li>👉 TTI &gt;&gt;&gt; FCP</li> <li>👉 Usually buffered</li> </ul>	<ul style="list-style-type: none"> <li>👉 TTI &gt; FCP</li> <li>👉 Limited streaming</li> </ul>	<ul style="list-style-type: none"> <li>👉 TTI &gt;&gt;&gt; FCP</li> <li>👉 No streaming</li> </ul>
Scales via:	Infra size / cost	build/deploy size	Infra size & JS size	JS size	JS size
Examples:	Gmail HTML, Hacker News	Docusaurus, Netflix*	<a href="#">Next.js</a> , <a href="#">Razzle</a> , etc	Gatsby, Vuepress, etc	Most apps