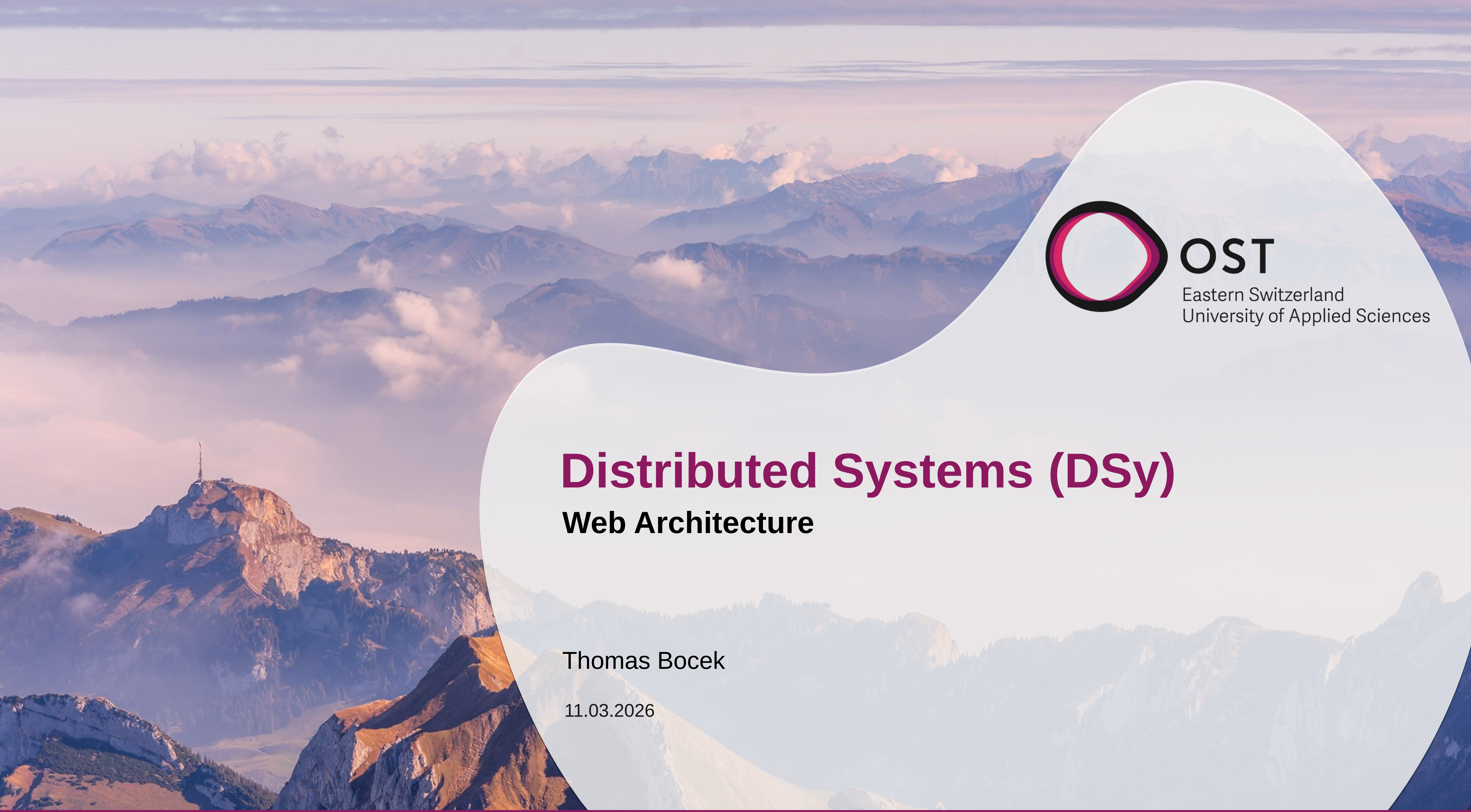


OST
Eastern Switzerland
University of Applied Sciences

Distributed Systems (DSy)

Web Architecture

Thomas Bocek

11.03.2026

Learning Goals

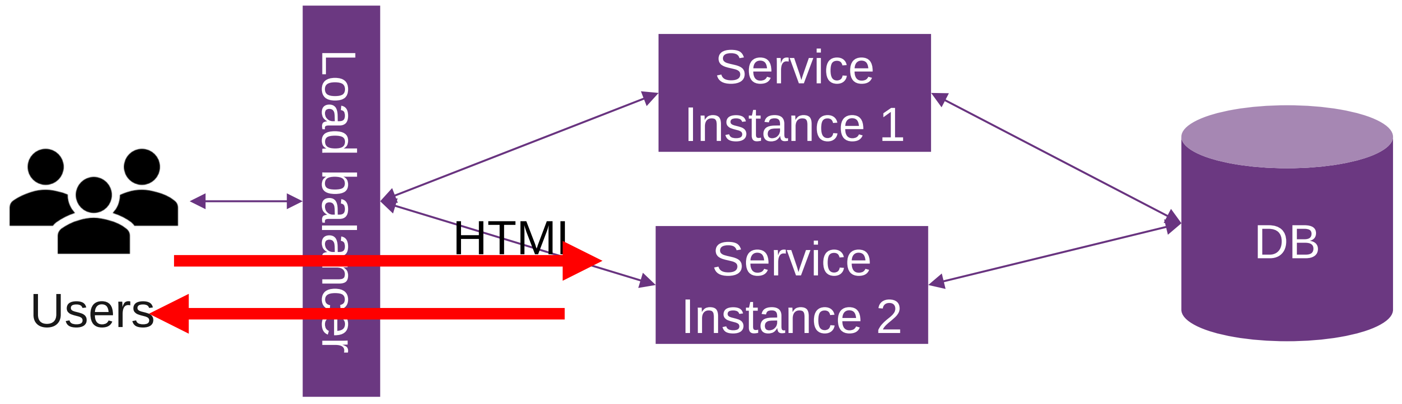
- Lecture 5
  - What are the options to build my challenge task?
  - What is currently “state-of-the-art”?
  - CORS

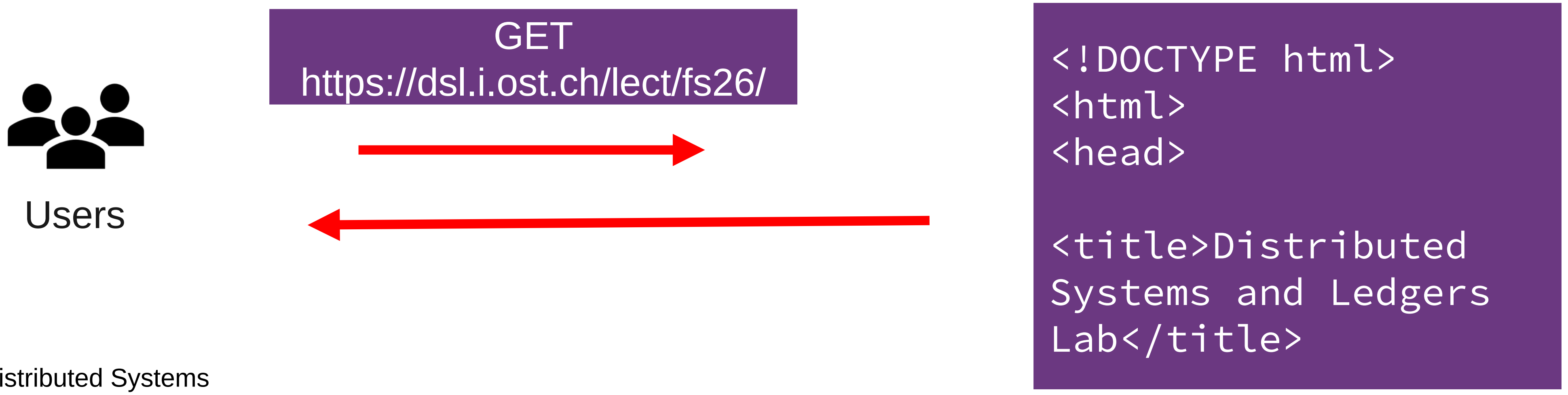
Server-Side Rendering

- “Classic” approach - “SSR”
- Server generates HTML/JS/CSS (dynamically)
  - User request: browser sends a request to the web server (server-side routing)
  - Server processing: server processes request by running server-side code (C#, Java, …)
    - May require data from database or other sources
    - Server-side code can use template engines or a framework to render the HTML
  - Response: Generate the appropriate HTML, CSS, and JavaScript for the requested page.
  - Browser rendering: browser receives response and renders page
- Advantage: SEO, immediate display
- Disadvantage: server rendering for every request (caching!), UI logic on the server
- Static site generation (SSG): pre-render HTML/CSS/JS: only once, regenerate if content changes
  - https://dsl.i.ost.ch → markdown to HTML

Server side rendering (SSR) Simple Example

- Request entire page





```
GET
https://dsl.i.ost.ch/lect/fs26/
```

```
<!DOCTYPE html>
<html>
<head>

<title>Distributed
Systems and Ledgers
Lab</title>
```

Single Page Application SPA / CSR

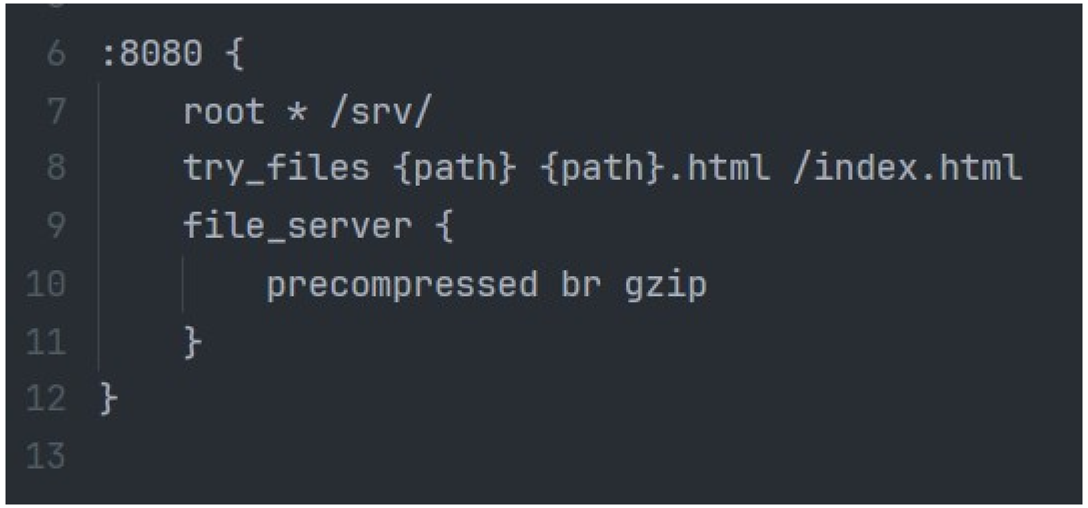
- Interactions occur within a single web page
- App-like experience: client page dynamically updates as user interacts
- Relies on JavaScript to update UI, typically:
  - Initial request: browser sends a request to receive initial (almost empty) HTML, JS, CSS
  - Initial response: server returns a single HTML file with references to CSS/JavaScript
  - Browser rendering: shows initial empty HTML file, with a spinner, then executes JavaScript, then shows UI

- User interactions: JavaScript manages the UI updates. Application does not require full page reloads. When you click a link in an SPA, instead of making a traditional HTTP request:
  - JavaScript intercepts the click event
  - Prevent default browser navigation
  - Update the URL using the History API
  - Render new content without requesting new HTML document, but may involve fetching data
- Fetching data: When the SPA needs to fetch or send data, communicates through APIs

Single Page Application SPA / CSR

- Use a framework: React, Angular, Vue
- Backend serves API requests only
- SEO only works if JavaScript is executed
  - Crawler gets JavaScript code, needs to execute, then it knows the content
    - Many corner cases (endless loops?)
- Good separation: UI in HTML/CSS/JS, backend in /api
- Client-side routing: SPAs for navigation
  - Server side routing? – default to index.html, as client side routing “inside” index.html

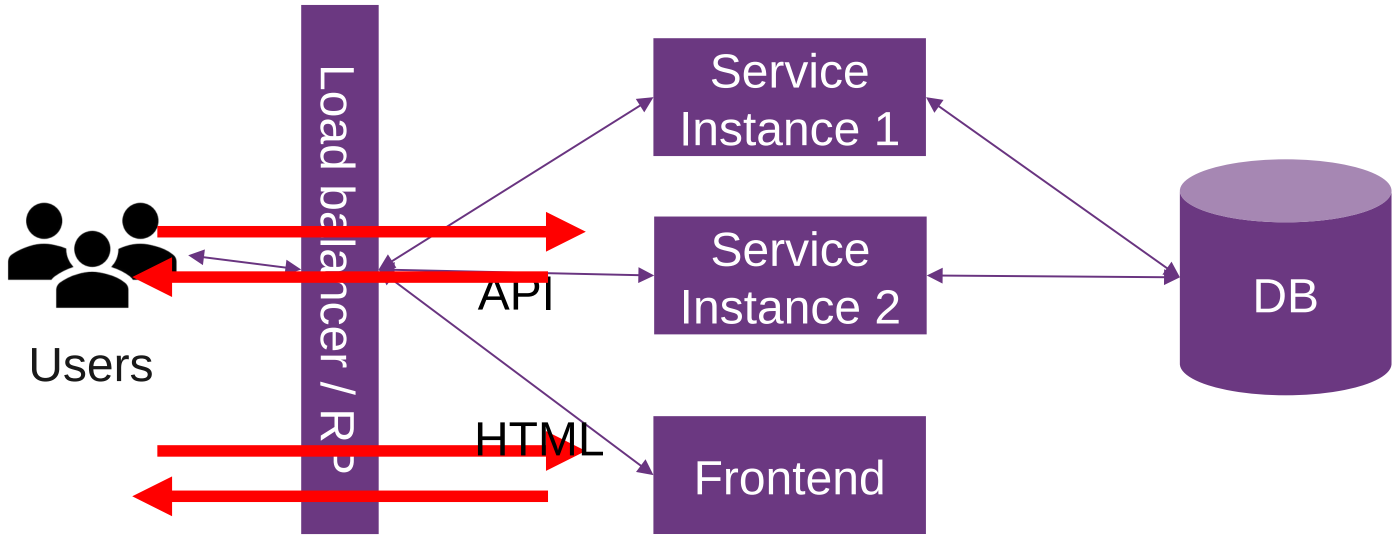
- Typical setup
  - / → index.html
  - /user → user.html
    - Alternatively: /user/index.html (not in config)
- Simplified Vue example (no build step)



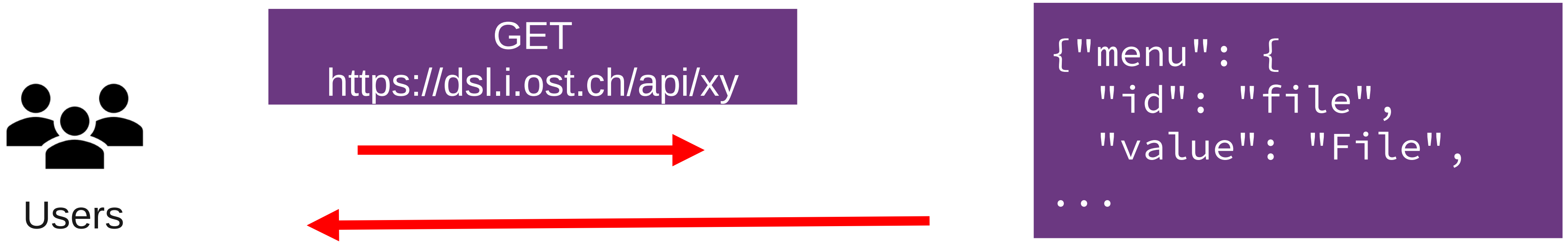
```
:8080 {
    root * /srv/
    try_files {path} {path}.html /index.html
    file_server {
        precompressed br gzip
    }
}

```

Simple Example



- Initial load: entire page
- Further requests: only updates partially



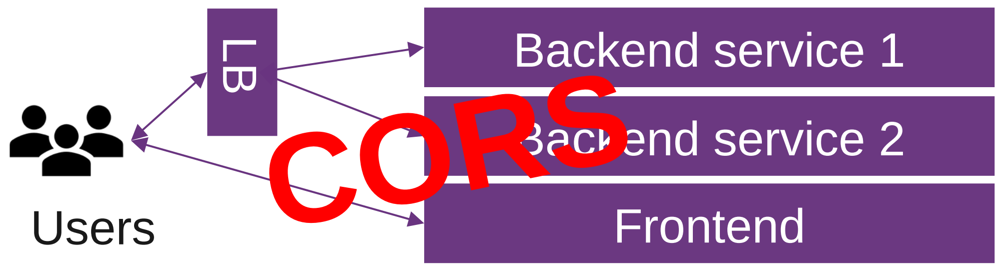
```
{"menu": {
  "id": "file",
  "value": "File",
...
```

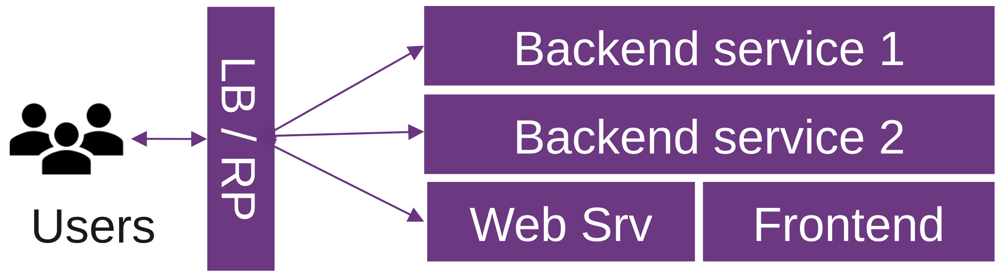
CORS

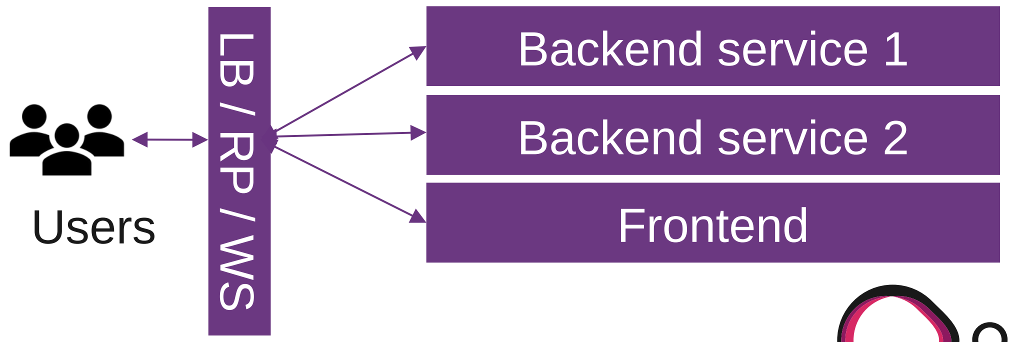
- CORS = Cross-Origin Resource Sharing
  - For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts (among others)
  - Mechanism to instruct browsers that runs a resource from origin A to run resources from origin B
- Solution: reverse proxy with webserver
  - Many solutions, caddy, vite/webpack dev server, nginx, mix

    → The client only sees the same origin for the API and the frontend assets
- "Workaround": Access-Control-Allow-Origin: https://…
  - For dev: Access-Control-Allow-Origin: *
  - Pre-flight requests
  - Golang: w.Header().Set("Access-Control-Allow-Origin", "*")

- Reverse proxy

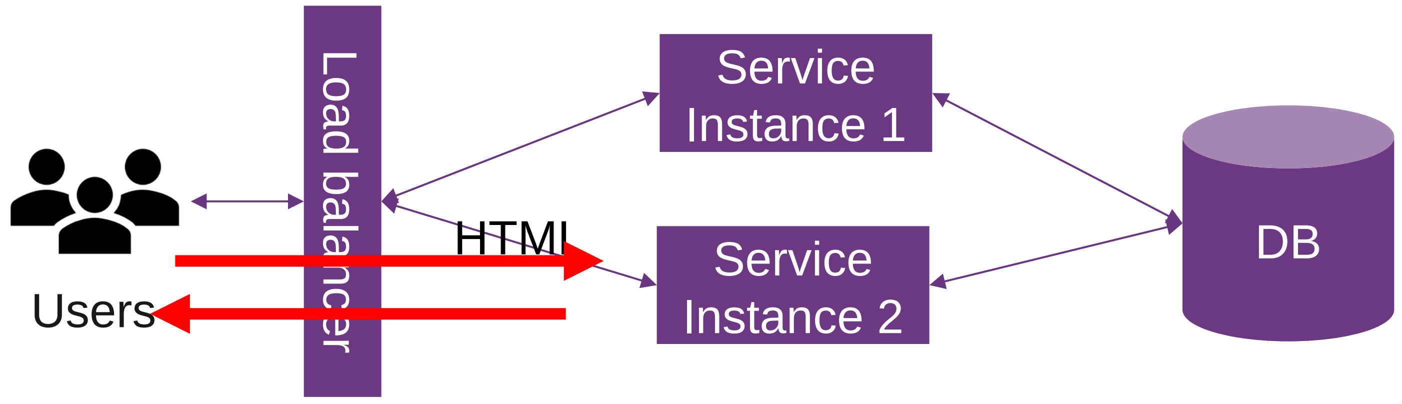


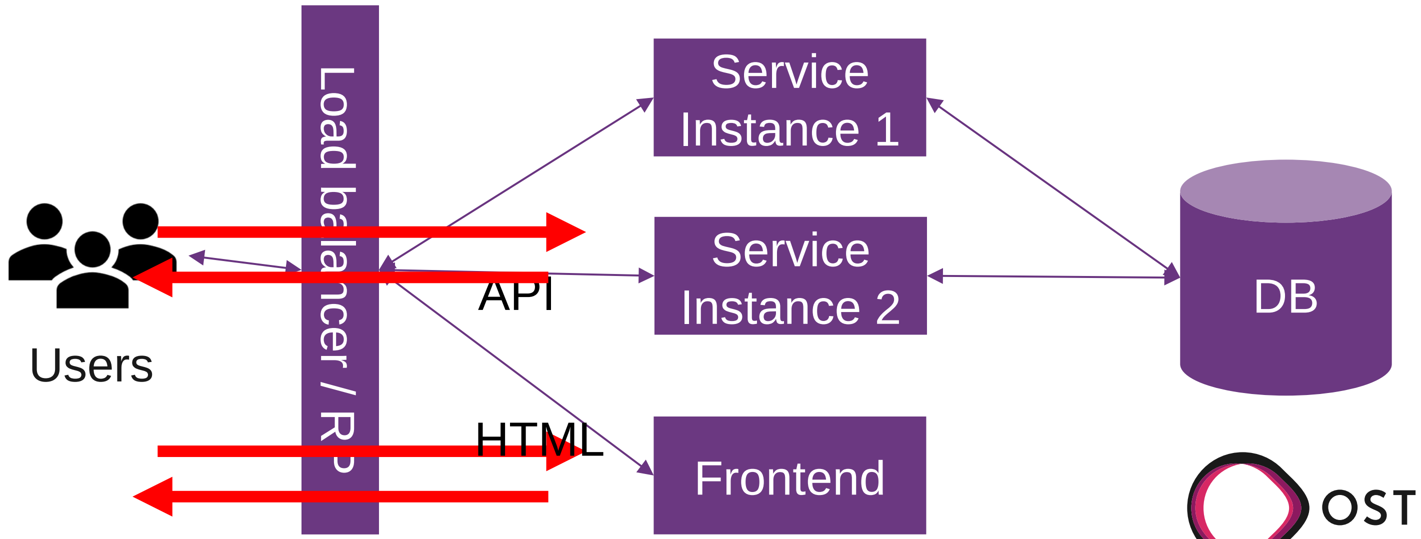




Architecture Comparison

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





Comparing SSR / CSR

- Performance Metrics
  - TTFB: Server response time
  - FCP: When content first appears
  - TTI: When page becomes fully interactive
  - Trade-off: SSR good at FCP/TTI, CSR requires full JS execution first
- SSR vs CSR Initial Load
  - SSR: visible and interactive immediately (no JS needed)
  - CSR: Must download, parse, execute JS before interactive
  - Post-load: CSR good: no page reloads for navigation, feels like desktop app
- CSR Advantages: lower server rendering load, API only serves JSON
- CSR Disadvantages
  - Bundle Size Problem: Large JS files, slow parse/execution, mobile may struggles
  - Slow initial load: white screen until JS executes
  - SEO Problem: Crawlers see empty HTML, need JS executing to read content
- Why CSR despite slower initial load? IMHO
  - Clean architecture (frontend/backend separation)

OST

CSR Improvements (2)

- Code splitting, lazy loading, hybrid approach with hydration
  - Hydration Problems
    - Duplicated work / complexity
    - Pre-rendering only: PrevelteKit
- Server Components (React)
  - Components render only on server, no JS sent to client
  - Reduces bundle size
- Server HTML fragments
  - htmx: server replies with HTML fragments
- Islands Architecture
  - Static HTML with interactive islands
  - Only islands ship JS - minimal JS by default (Astro)
- Streaming SSR
  - Send HTML in chunks as ready
  - Browser renders earlier - improves perceived performance (Qwik)
- Edge Rendering
  - Render closer to user geographically at CDN edge locations (Cloudflare Workers, Vercel Functions)