From Script Tags to Edge Functions

Why web development is so complicated

Ansgar Hoyer

- started Web-Development in 2016
- Software Engineer at AOE since 2023
- focus on JavaScript and TypeScript Ecosystem
- lecturer at HSRM

AnsgarH1

Ansgar Hoyer

slides.techtalk.ansgar.app

The Early Days of JavaScript

JavaScript's Dual Origins

- **1995**: Netscape creates JavaScript (Brendan Eich)
  - Created in just 10 days
  - Originally named "Mocha", then "LiveScript"

- **1996**: Microsoft releases JScript for Internet Explorer
  - A reverse-engineered version of JavaScript
  - Similar but with intentional differences

*picture of Bill Gates as reference

The Browser Wars

Netscape Navigator

- First commercial browser
- JavaScript as a competitive feature
- Dominant in early web

Internet Explorer

- Microsoft's answer
- Bundled with Windows
- JScript implementation
- Eventually won with IE6

Different APIs, different behaviors, different bugs...

Same code would work differently (or not at all) across browsers!

Birth of ECMAScript

- **1997**: JavaScript standardized as ECMAScript (ECMA-262)
  - European Computer Manufacturers Association
  - Netscape submitted JavaScript for standardization
  - Goal: Create a vendor-neutral, cross-platform language standard
- **Early versions**:
  - ES1 (1997) - Initial standardization
  - ES2 (1998) - Editorial changes
  - ES3 (1999) - Regular expressions, try/catch, etc.
  - ES4 - Abandoned after disagreements
  - ES5 (2009) - JSON, strict mode, many improvements

The Dark Ages (2000-2009)

- Internet Explorer dominance (IE6)
- Stagnation in browser innovation
- **JavaScript libraries emerge to handle cross-browser issues**:
  - jQuery (2006)
  - Prototype.js
  - MooTools
  - Dojo Toolkit

```
// jQuery example - hiding cross-browser complexity
$("#myButton").click(function() {
  $(".elements").fadeIn(500).addClass("active");
});
```

Monolithic Web Frameworks emerged

- Server-side rendered pages with minimal JavaScript
- Tightly coupled frontend and backend code
- Full-stack frameworks handling routing, templates, and database
- "LAMP Stack era"
- typically poor user experience

Monolithic Web Framework Performance

Renaissance: Modern JavaScript

ES6/ES2015: The biggest update to the language

- Arrow functions, classes, modules, promises, etc.
- Annual release cycle begins (ES2016, ES2017)

New browsers and engines:s

- Chrome with V8 (2008)
- Firefox with SpiderMonkey
- Browser competition drives JS engine performance

```
// ES6 features
const add = (a, b) => a + b;

class Person {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    return `Hello, I'm ${this.name}`;
  }
}

// Modules
import { Component } from './component';
```

The rise of cloud computing

- Software moved from on-premise to the cloud
- Cloud providers offer a wide range of services
- Serverless computing became a thing
- CDNS became widely available, Hosting static files became basically free
- Cloud Platforms reselling AWS services but with a nicer UI emerged

Static Web Pages returned

- only HTML, CSS, JS
- no complex webserver needed, only static file serving
- CDN can serve the files
- Tools like Github-Pages or Netlify-Drop made hosting easier than ever

Netlify Drop

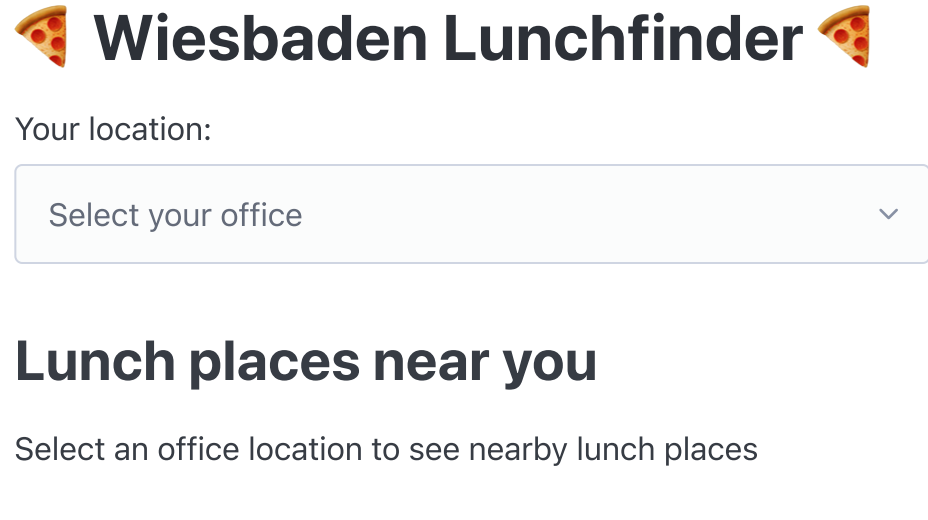
Demo Time!

Demo 1: Vanilla JavaScript

Traditional web development with HTML, CSS, and JavaScript No build tools, no frameworks*

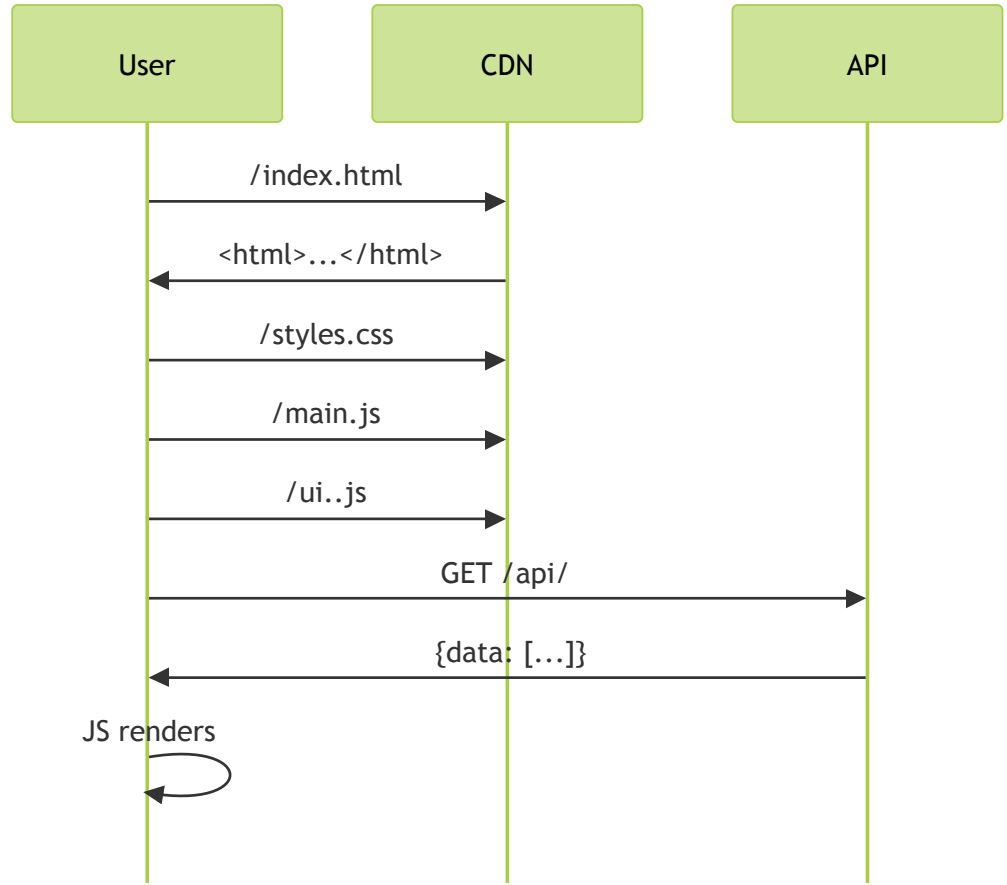
Code: Demo1

Web: https://demo-01.techtalk.ansgar.app



Demo 1:

Request Flow



Demo 1: Vanilla JavaScript

Demo 1: Vanilla JavaScript

Web Performance

The Transpiler & Bundler Era

The Compatibility Problem

- Modern JavaScript features, but older browsers don't support them
- Businesses need to support legacy browsers (IE11 until recently)
- Can't use latest language features directly

How do we write modern code but still support older browsers?

Enter: Transpilers

Transpilers: Transform modern JavaScript into backward-compatible versions

- **Babel**: The most popular JavaScript transpiler
  - Write ES2015+ code
  - Babel converts it to ES5 for wider browser support

```
// Modern JavaScript (input)
const greet = (name) => `Hello, ${name}!`;
let [first, ...rest] = [1, 2, 3, 4];

// Transpiled to ES5 (output)
"use strict";
var greet = function greet(name) {
  return "Hello, ".concat(name, "!");
};
var _ref = [1, 2, 3, 4],
    first = _ref[0],
    rest = _ref.slice(1);
```

Module Systems & Bundlers

Problem: Browser support for modules was inconsistent

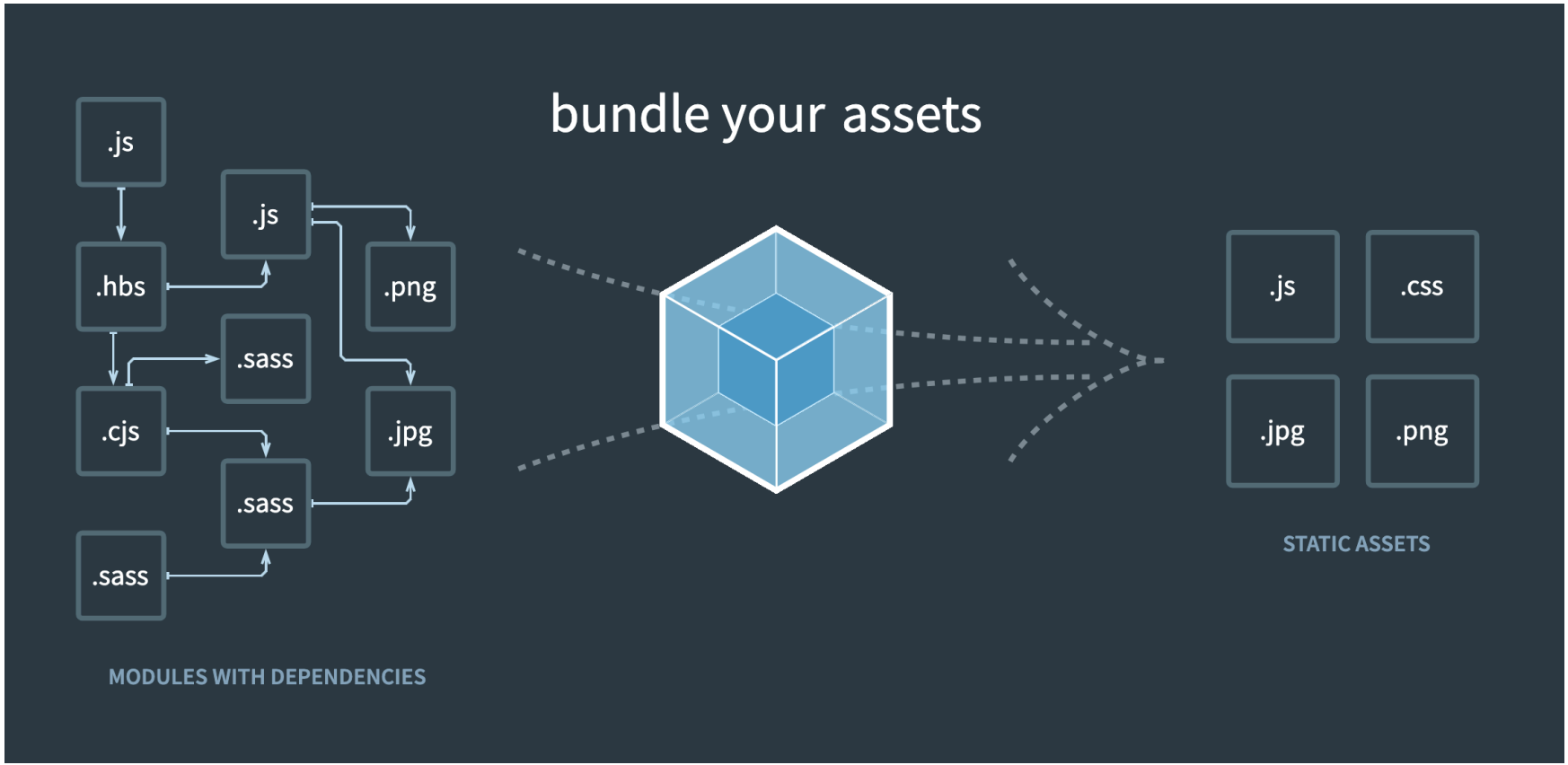
Module formats evolved:

- CommonJS (Node.js): **require()** and **module.exports**
- AMD (RequireJS): For browsers, async loading
- UMD: Universal Module Definition (works in both)
- ES Modules: The official standard (**import/export**)

Bundlers: Package multiple modules into fewer files

- **Webpack**: Most popular bundler
- **Rollup**: Focused on ES modules and tree-shaking
- **Parcel**: Zero-config bundler

Webpack


bundle your assets
.js
.hbs
.cjs
.sass
.sass
.js
.png
.sass
.jpg
MODULES WITH DEPENDENCIES
.js
.css
.jpg
.png
STATIC ASSETS

TypeScript: Type Safety for JavaScript

- **Released by Microsoft in 2012**
- **Key benefits**:
  - Static type checking
  - Better IDE support
  - Self-documenting code
  - Catches errors at compile time

```typescript
// TypeScript example
interface User {
  id: number;
  name: string;
  email: string;
  role: 'admin' | 'user' | 'guest';
}

function sendEmail(user: User, message: string): Promise<boolean> {
  // Type-safe implementation
  return fetch(`/api/email/${user.id}`, {
    method: 'POST',
    body: JSON.stringify({ message })
  }).then(res => res.json());
}
```

JavaScript Runtimes

Each runtime provides different APIs and capabilities!

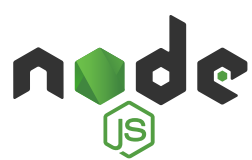
Browser

- The original JS environment
  - V8 (Chrome/Edge)
  - SpiderMonkey (Firefox)
  - JavaScriptCore (Safari)



Server-side

- JavaScript beyond the browser
  - Node.js (2009) - Server-side JavaScript
  - Deno (2018) - Secure by default
  - Bun (2023) - Performance focused




Package Managers: Dependency Hell

npm (2010): Node Package Manager

- Revolutionized code sharing
- *node_modules* became infamous

Yarn (2016): Faster, more reliable

- Introduced lockfiles
- Parallel installations

pnpm (2017): Disk space efficient

- Symlinks to a central store
- Strict dependency resolution

The average project now has hundreds of dependencies!

Build Systems: Taming the Complexity

The Configuration Nightmare

- Complex configuration files
- Countless plugins
- Tedious setup
- Slow builds

Development became about configuring tools rather than writing code!

Modern Build Systems

- **Parcel** (2017)
  - Zero-config bundler for any project
- **Vite** (2020)
  - Lightning-fast dev server using ES modules
- **Turbopack** (2022)
  - Announced as "successor to Webpack"
- **Rspack, Rsbuild** (2022)
  - Another successor to Webpack
- 🪦 RIP : Create React App, Vue CLI

```
# Complex setups got abstracted by simple cli-commands
npm create vite@latest -- --template react-ts
# or
npm create parcel react-client my-parcel-app
# or
npm create rsbuild@latest
# or with Next.js
npx create-next-app@latest
```

* please don't actually use npm anymore

Vite: Revolution in DX

- Developed by Evan You (Vue creator)
- **Key innovations**:
  - Uses native ES modules in development
  - Only bundles for production
  - Incredibly fast hot module replacement
  - Framework-agnostic

```
// vite.config.js - Simple configuration
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
})
```

Linting & Formatting

ESLint

- Catch bugs and enforce conventions - Pluggable architecture - Hundreds of rules available

Prettier

- Opinionated code formatting
- End debates about code style
- Consistent formatting across teams

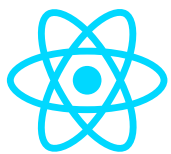
Husky & lint-staged

- Enforce at commit time
- Pre-commit hooks
- Only check changed files

More tools to learn, configure, and maintain...

even with new technologies, writing vanilla JavaScript has still its limits...

Single-Page Applications



Single-Page Applications: Component Composition

Building Blocks

- **Reusable Components**
  - Self-contained UI pieces
  - Props for configuration
  - Events for communication
- **Component Hierarchy**
  - Parent-child relationships
  - Nested components
  - Component trees

Benefits

- **Code Reusability**
  - DRY principle
  - Consistent UI patterns
- **Maintainability**
  - Isolated functionality
  - Easier testing
  - Clear responsibilities
- **Development Efficiency**
  - Component libraries
  - Faster UI development
  - Team collaboration

Single-Page Applications: New Paradigms

New Concepts

- **Client-side routing**
  - Navigation without page reload
  - History API manipulation
- **State management**
  - Redux, Vuex, MobX, Pinia, Zustand
  - Complex client-side data handling
- **Virtual DOM**
  - Efficient UI updates
  - DOM diffing algorithms

Architecture Changes

- **Frontend-backend separation**
  - APIs instead of server templates
  - JSON over HTML responses
- **Rich client experiences**
  - Animations and transitions
  - Instant feedback
- **New challenges**
  - SEO difficulties
  - Initial load performance
  - JavaScript dependency

Vue.js

"The Progressive JavaScript Framework"

- **Released by Evan You in 2014**
- **Key features**:
  - Incremental adoption
  - Reactive data binding
  - Template syntax
  - Component system
  - Single File Components

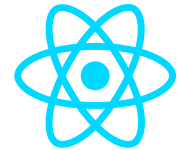
```vue
<script setup>
import { defineProps } from 'vue';

defineProps({
  user: User,
});

const followUser = (userId) => {
  console.log(`Following user: ${userId}`);
};
</script>
<template>
  <div class="profile">
    <img :src="user.avatar" :alt="user.name" />
    <h2>{{ user.name }}</h2>
    <p>{{ user.bio }}</p>
    <button @click="followUser(user.id)">Follow</button>
  </div>
</template>
```

React

" The library for web and native user interfaces"



- **Released by Facebook in 2013**
- **Key innovations**:
  - Component-based architecture
  - Virtual DOM for performance
  - Declarative UI programming
  - Unidirectional data flow

```
type UserProfileProps = {
  user: User
};

function UserProfile({ user }: UserProfileProps) {
  const followUser = (userId) => {
    console.log(`Following user: ${userId}`);
  };
  return (
    <div className="profile">
      <img src={user.avatar} alt={user.name} />
      <h2>{user.name}</h2>
      <p>{user.bio}</p>
      <button onClick={() => followUser(user.id)}>
        Follow
      </button>
    </div>
  );
}
```

Demo 2: Vue.js Single-Page-Application

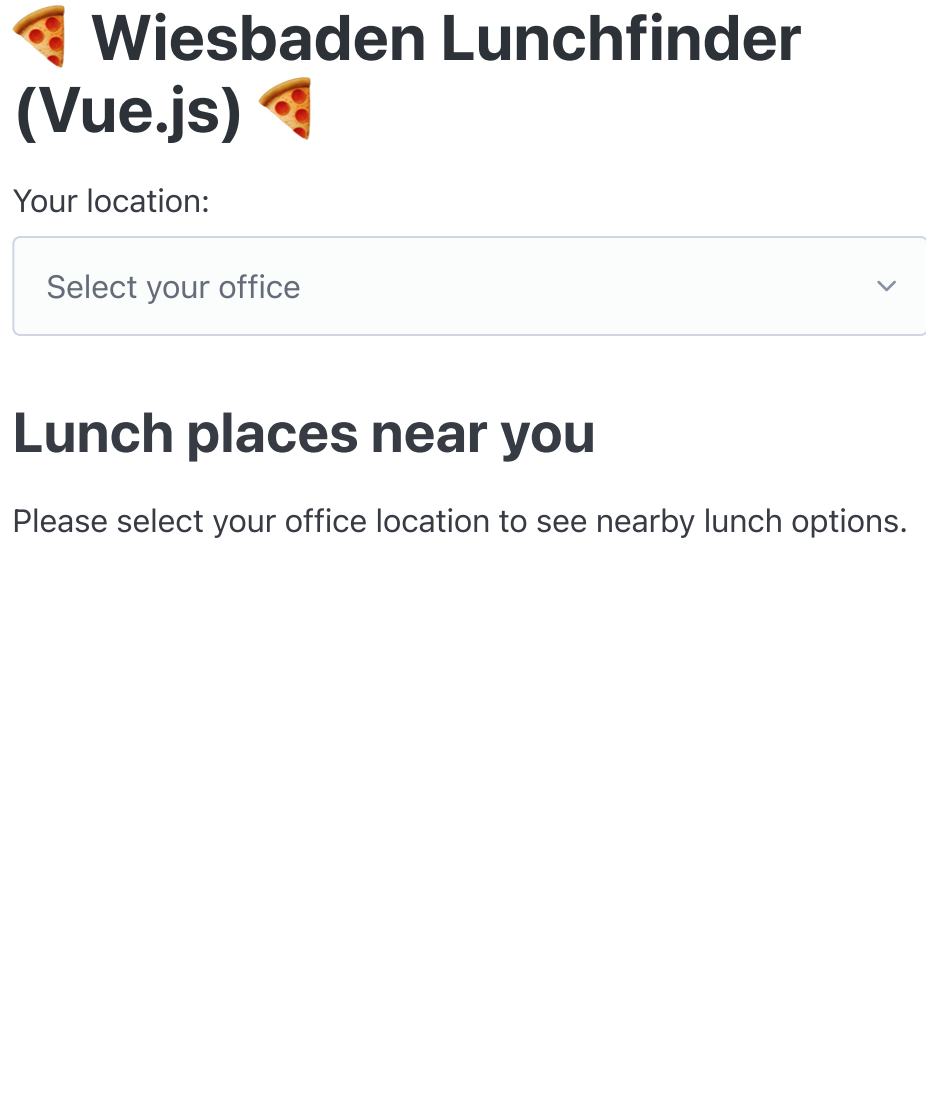
Modern single-page application with a fast build system Client-side rendering with instant HMR

Demo 2: Vue.js SPA

Client-side rendered single-page application
Fast development with Vite and Hot Module Replacement

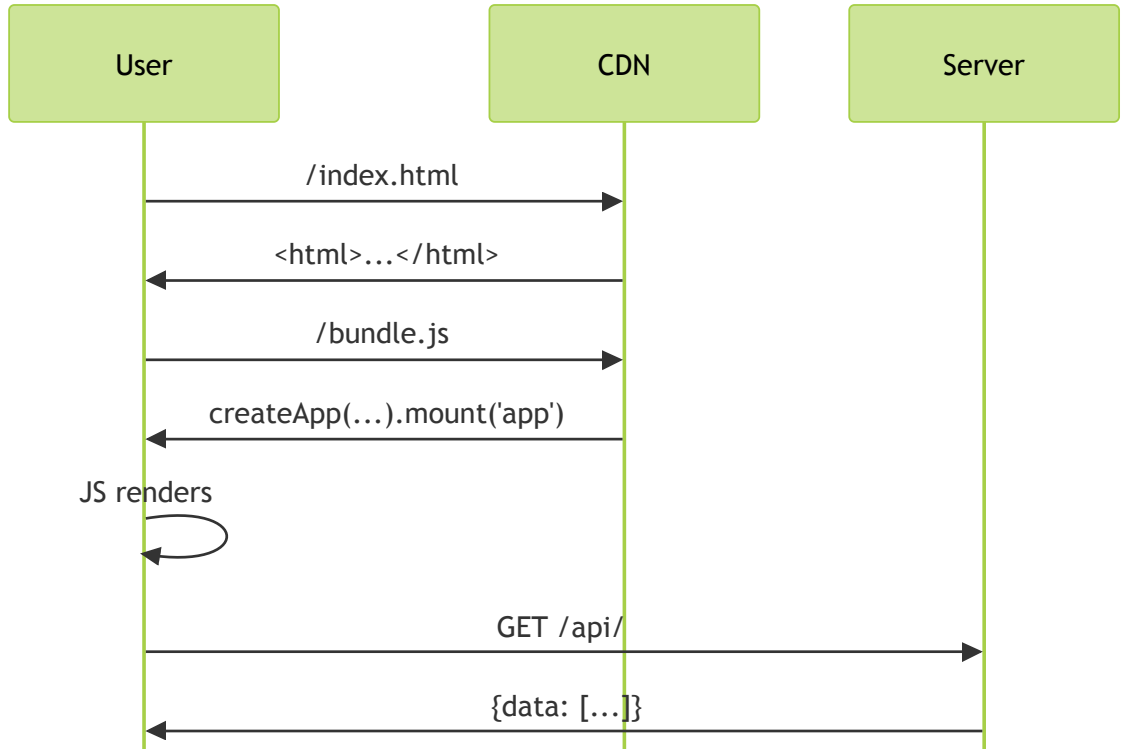
Code: Demo2

Web: demo-02.techtalk.ansgar.app



Demo 1:

Request Flow



Demo 2: Vue.js SPA

Demo 2: Vue.js SPA

Web Performance

SPAs are slower than classical MPAs?

The Return of the server

- HTML generation is done on the server again
- but: we still use the same frameworks and libraries (React, Vue, Svelte, …)
- the server sends the HTML to the client
- the client then "hydrates" the static HTML with the JS bundle and continues as SPA
- known as Server-Side Rendering (SSR)

SSR Performance

Demo 3: React SSR App

- needs Webserver, that renders the React App on the server
- sends the HTML to the client
- client "hydrates" the HTML with the JS bundle and continues as SPA

Code: Demo3

Bun to the rescue!

How can we further optimize the performance?

Improve rendering performance:

- alread load data from api during server render
- run the "server render" already during build time (known as Static Site Generation)
- cache the rendered output to serve it immediately
- stream the html as it gets generated
- break up the side into static and dynamic parts

Improve infrastructure performance:

- we already put the static files near the user by using a CDN
- why not move the api near the user?
- why not move the database near the user?

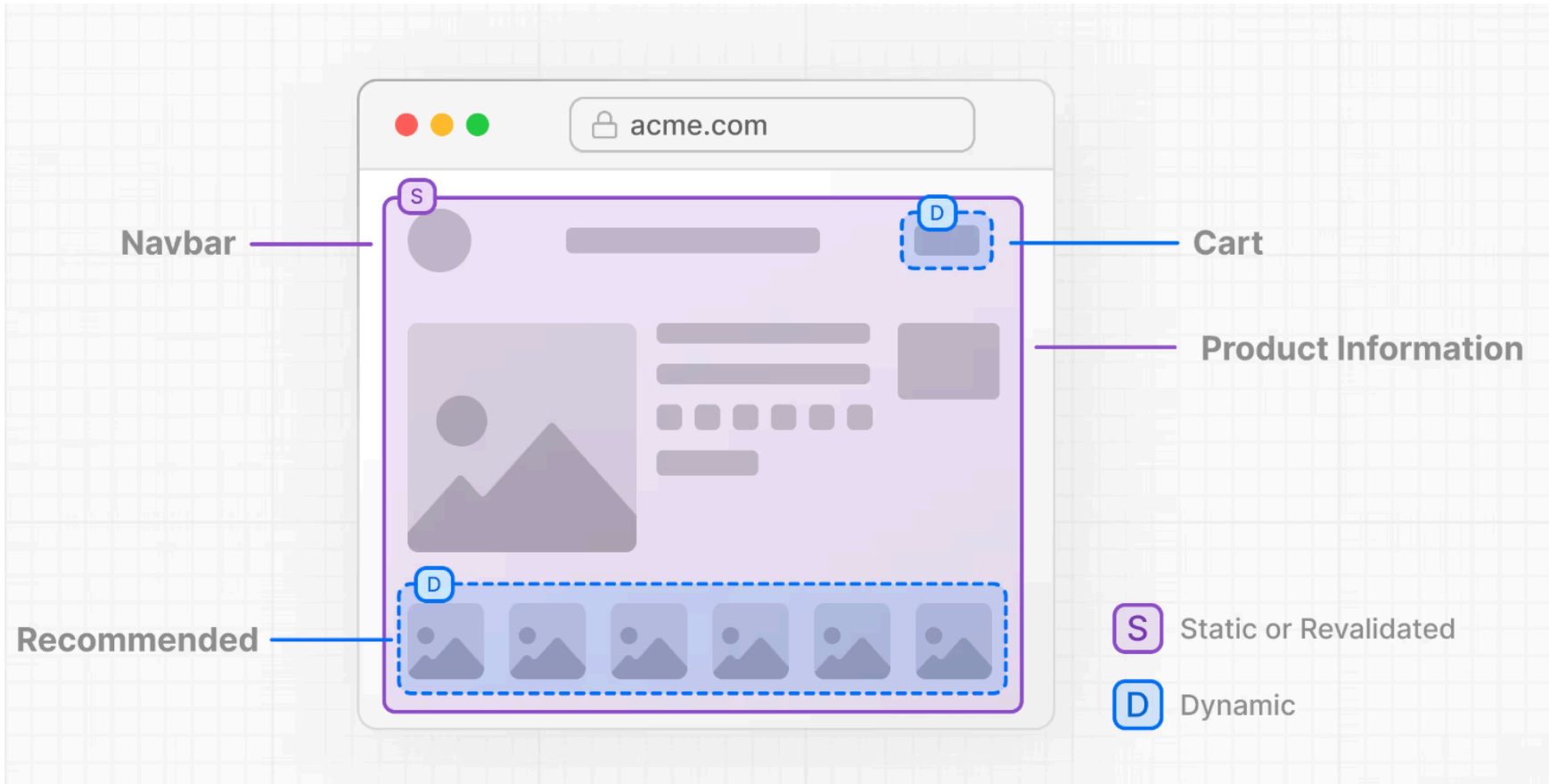
Improving rendering performance

The elefant in the room:

- Runs the web server
- provides the build system
- automatically does all of the performance optimizations
- provides features like routing, image optimization, server-side loading, etc.

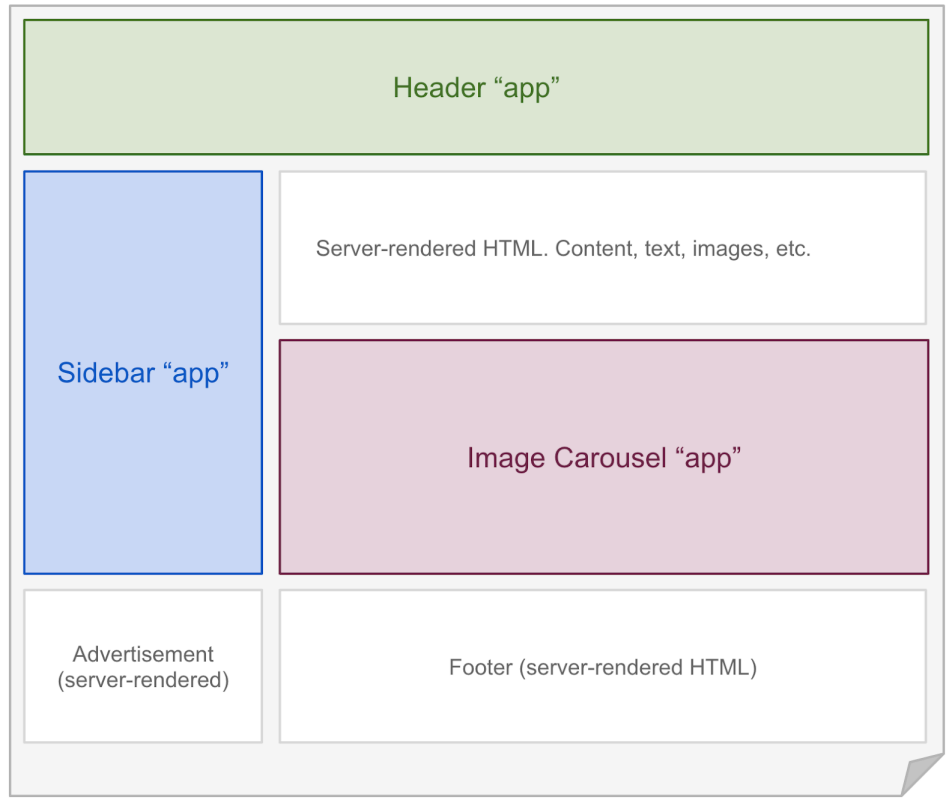
But:

Partial prerendering



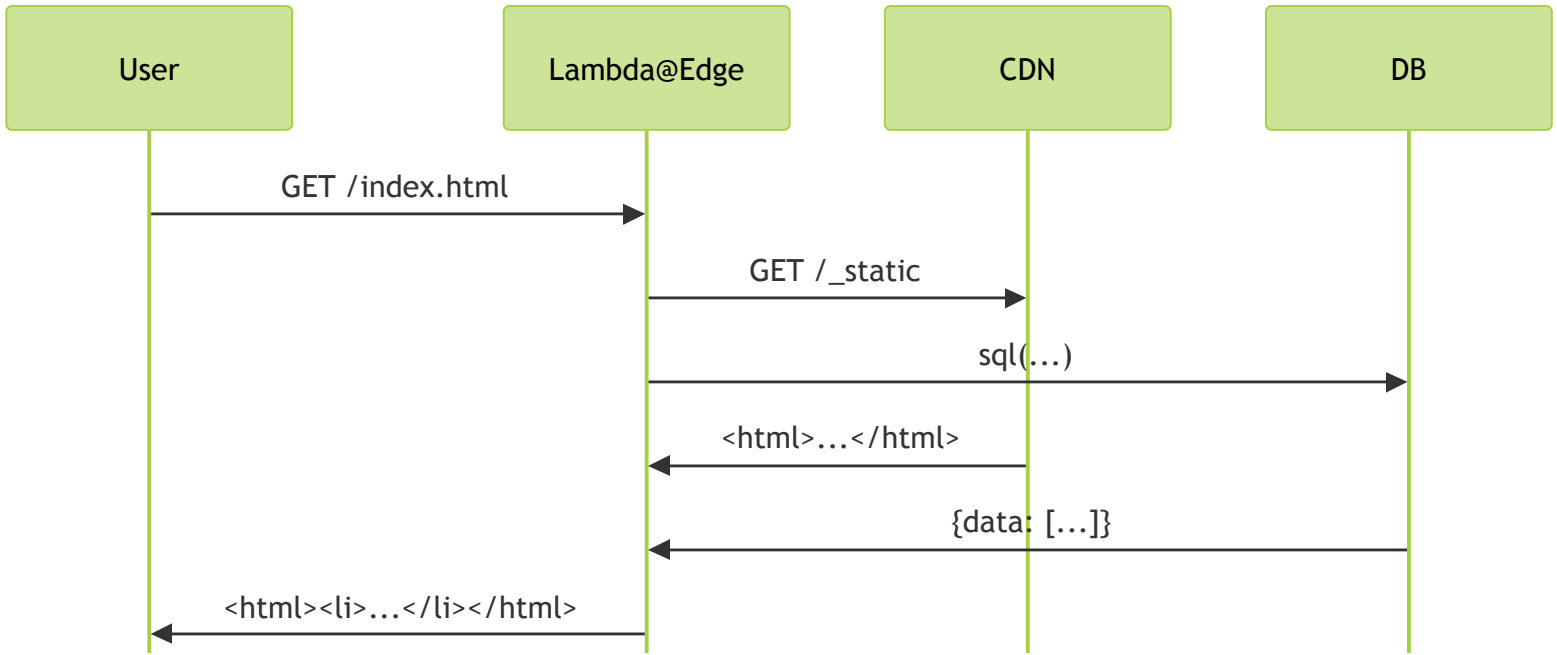
Astro: Islands Architecture

astro



Improving infrastructure performance

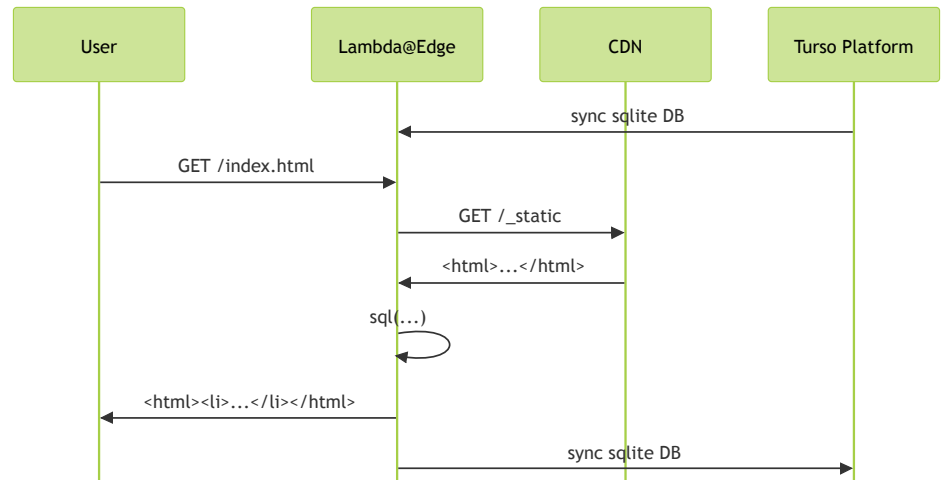
Getting the most out of SSR (with PPR)



Putting the database near the user

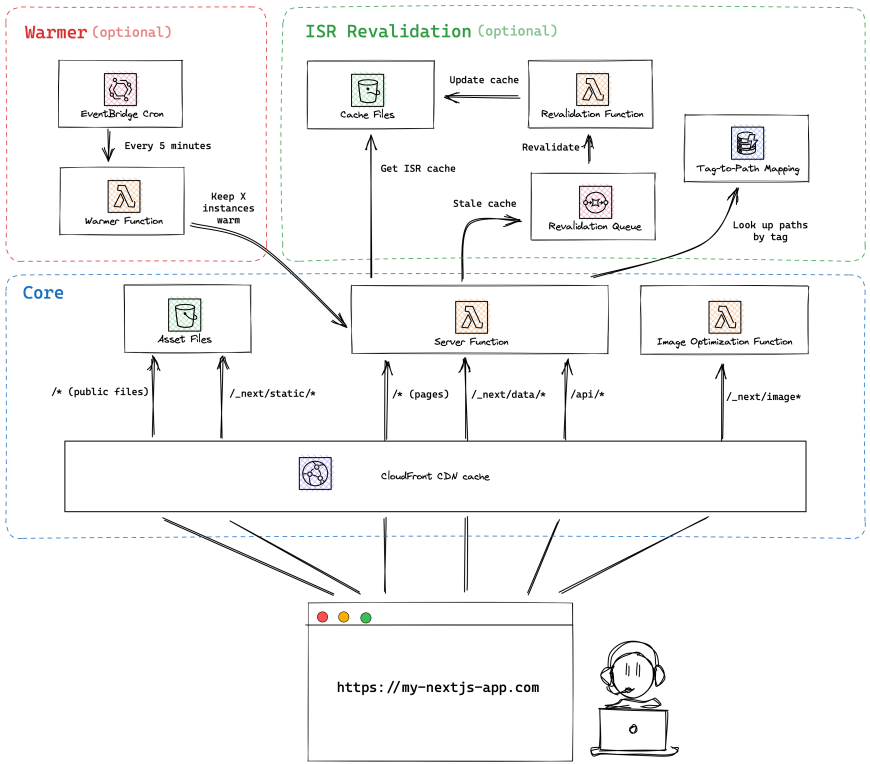
TURSO

- Turso is a Database Platform that utilizes SQLite
  - allows for embedded database replicas inside the Applications
  - allows for concepts like per tenant databases, branching, etc.



How to run Frontend Frameworks

How to run Next.js **serverless** without Vercel?



- Next still can get run in a Docker Container
- Only other current options are Cloudflare and Netlify
- running in Docker Container is still possible, but performance varies
- Vercel uses a complex setup of Lambdas, K8s, Cloudflare and edge computing to archive the best performance

What next?

Topics for next time

- React Server Components (and new React Frameworks)
- New JS Tools in Rust
- JS Fullstack Frameworks (Nuxt, Solid-Start, SvelteKit, …)
- The mess of all the JS Runtimes that currently exist
- Testing setups
- We somehow didn’t even talk about CSS and all the other styling solutions

Will this get any easier?

- Modern tools like Bun and Vite simplify the tooling setup
- new Libraries make building performant apps easier than ever
- Infrastructure problems will eventually get solved as well

Thank you!

Slides can be found at https://slides.techtalk.ansgar.app

- Demos:
  - https://demo-01.techtalk.ansgar.app
  - https://demo-02.techtalk.ansgar.app
- Slides: https://slides.techtalk.ansgar.app
- Lunchfinder API: https://api.techtalk.ansgar.app/api/ui
- Code, Slides and Demos: https://github.com/AnsgarH1/techtalk-wiesbaden-21-05-25