Frontend Foundations

> IE University - BCSAI - SDD - 2025

Table of Contents

1. The Foundations — HTML, CSS, and JavaScript
2. How Web Interfaces Are Built from Components
3. The Role of Frontend Frameworks (React, Vue, Angular, Svelte)
4. How Applications Manage Complexity Using State Management
5. How Routing Simulates Navigation in Single-Page Applications
6. How Browsers and Servers Communicate Through APIs
7. The Role of Build Tools (Webpack, Vite, Babel)
8. Why TypeScript Has Become a Standard
9. How Testing Ensures Reliability
10. How Performance Is Optimized
11. How Applications Are Packaged and Deployed
12. Modern Frontend Approaches: SSR, SSG, and Islands Architecture
13. Bibliography

---

1. The Foundations — HTML, CSS, and JavaScript

Modern web development begins with three foundational technologies that define the structure, presentation, and behavior of web interfaces.

HTML (Hypertext Markup Language)

HTML describes the **structure** and **semantic meaning** of content. It defines elements such as headings, links, images, buttons, and form fields.

Key characteristics:

- **Declarative**, not procedural.
- Describes *what* elements exist, not how they behave.
- Maintains meaning and accessibility for browsers, assistive technologies, and search engines.

Example:

```
<button>Submit</button>
```

Defines the presence of a button, but does not style or animate it.

CSS (Cascading Style Sheets)

CSS describes **presentation**—colors, typography, layout, spacing, animations.

Key concepts:

- **Selectors** match HTML elements.
- **Cascade** determines how rules override each other.
- **Box model** determines size and spacing.
- **Layout systems** such as Flexbox and Grid control placement.

Example:

```
button {
  background: blue;
  color: white;
  padding: 0.5rem 1rem;
}
```

JavaScript

JavaScript adds **behavior** and **logic**.

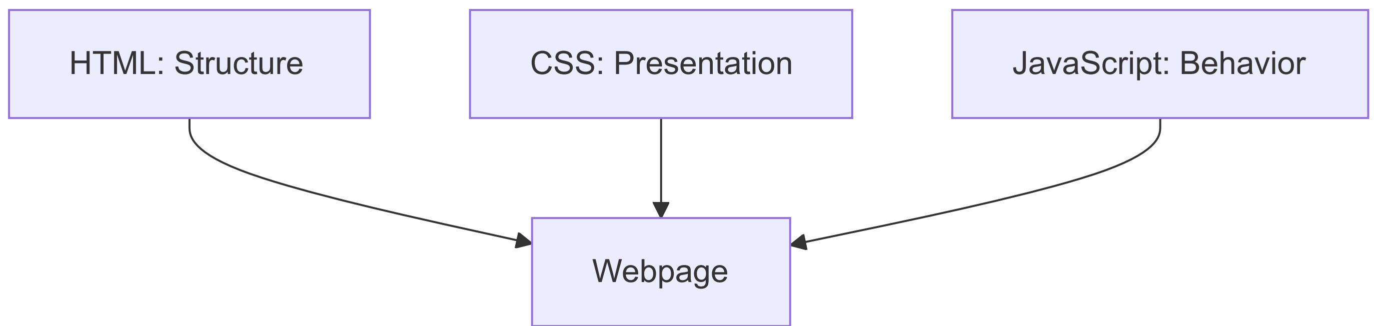
Responsibilities include:

- Handling clicks and user interactions
- Dynamically updating the DOM (Document Object Model)
- Retrieving and sending data via APIs
- Implementing application logic and state

Example:

```
document.querySelector("button").addEventListener("click", () => {
  alert("Submitted!");
});
```

Relationship Between the Three



HTML provides the skeleton, CSS provides the skin and layout, and JavaScript provides motion and interactivity.

---

2. How Web Interfaces Are Built from Components

As web applications grew in complexity, developers needed a way to break interfaces into smaller, reusable units. This led to the **component-based architecture** that modern frontend frameworks adopt.

What Is a Component?

A **component** is a self-contained UI unit that encapsulates:

- **Structure** (HTML or template)
- **Styles** (CSS or scoped CSS)
- **Behavior** (JavaScript logic)

Examples of components:

- A navigation bar
- A user profile card
- A product listing tile
- A modal dialog

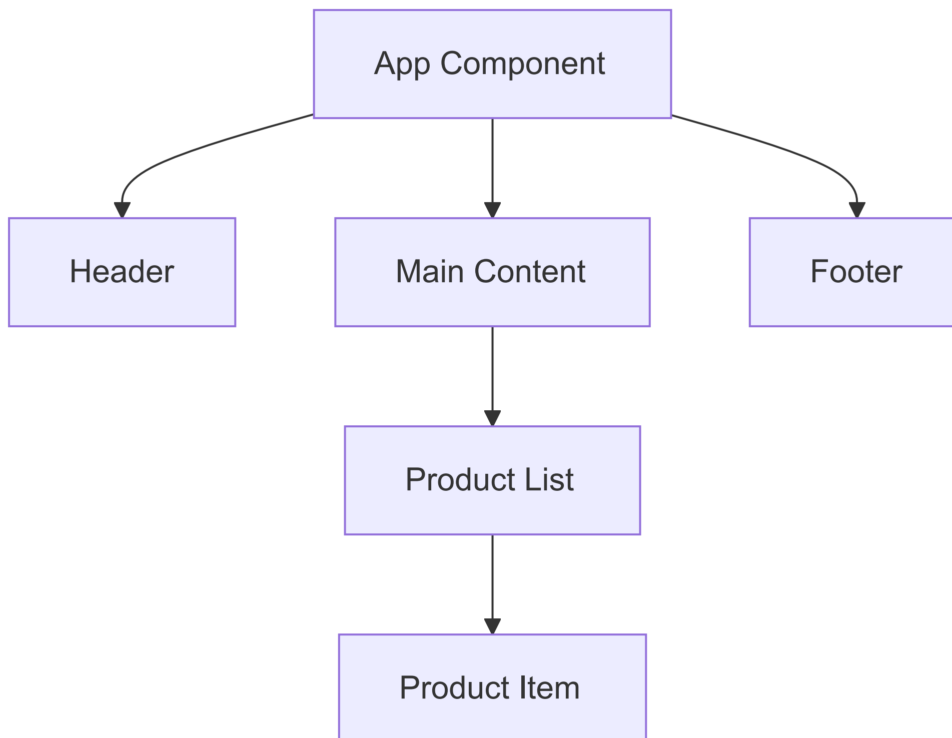
Why Components?

Components solve problems of scalability and maintainability:

- Reusability: Write once, reuse many times.
- Encapsulation: Internal logic and styles don’t leak.
- Composability: Components can be combined to form larger interfaces.

Component Hierarchies

Applications are built as trees of components.



In this structure:

- Changing or replacing **Product Item** does not affect the rest of the interface.
- **State** and **data** flow downward in the hierarchy (top → bottom).

UI Rendering and the DOM

Webpages are represented in the browser by the **DOM** (Document Object Model). When components update, they cause changes to the DOM.

However, frequent or inefficient DOM updates can be slow. This led to optimizations in frameworks discussed next.

---

3. The Role of Frontend Frameworks (React, Vue, Angular, Svelte)

Example Component Comparison

React

```
function Button({ label }) {
  return <button>{label}</button>;
}
```

Vue

```
<template>
  <button>{{ label }}</button>
</template>
<script>
export default { props: ['label'] };
</script>
```

Angular

```
@Component({
  selector: 'app-button',
  template: `<button>{{ label }}</button>`
})
export class ButtonComponent {
  @Input() label!: string;
}
```

Svelte

```
<script>
  export let label;
</script>
<button>{label}</button>
```

Modern frameworks simplify building dynamic interfaces by abstracting DOM management and providing structure.

React

- Uses **JSX** to define UI as functions of state.
- Employs a **Virtual DOM** to optimize updates.
- Encourages a **unidirectional data flow**.

```
function Counter() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count + 1)}>{count}</button>;
}
```

Vue

- Uses a template-based syntax.

- Provides **reactive data binding** out of the box.
- Easier to learn due to clear separation of concerns.

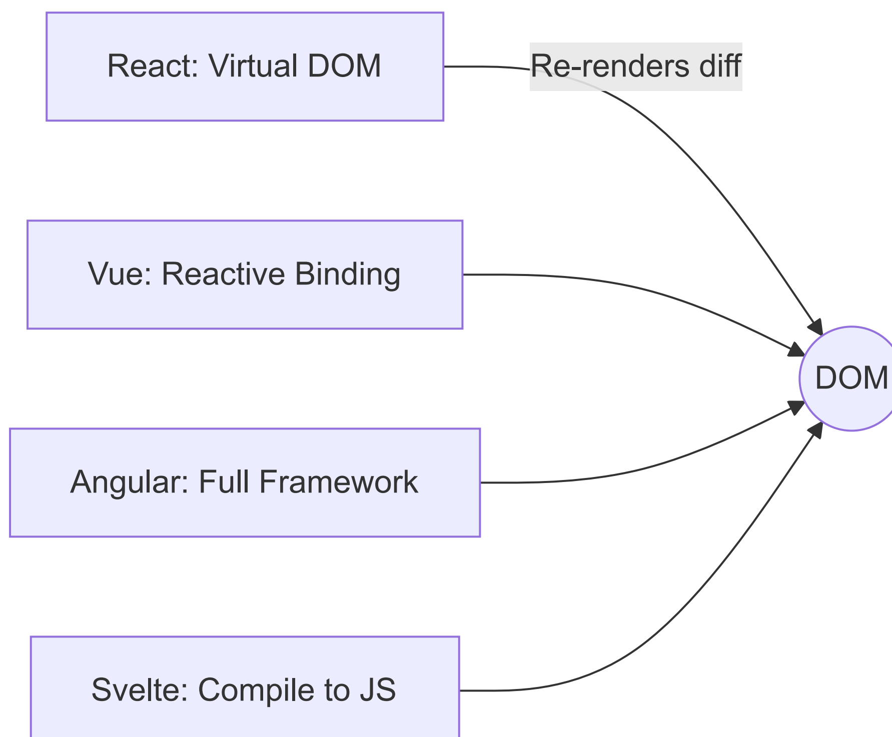
Angular

- A **full framework** including routing, HTTP utilities, and form management.
- Uses TypeScript by default.
- Based on a **hierarchical dependency injection** system.

Svelte

- Rather than using a Virtual DOM, Svelte compiles components to efficient JavaScript.
- Reduces browser overhead.

Comparing Conceptual Approach



Each framework aims to reduce **manual DOM manipulation** and provide consistency as applications scale.

---

4. How Applications Manage Complexity Using State Management

As applications grow, components need to share data. Managing state correctly prevents bugs and inconsistencies.

Types of State

Type	Description	Example
Local State	Belongs to one component	Form field value
Shared State	Needed by multiple components	Current user info
Server State	Fetched from backend APIs	Product list from DB
UI State	Describes what the UI is displaying	Which modal is open

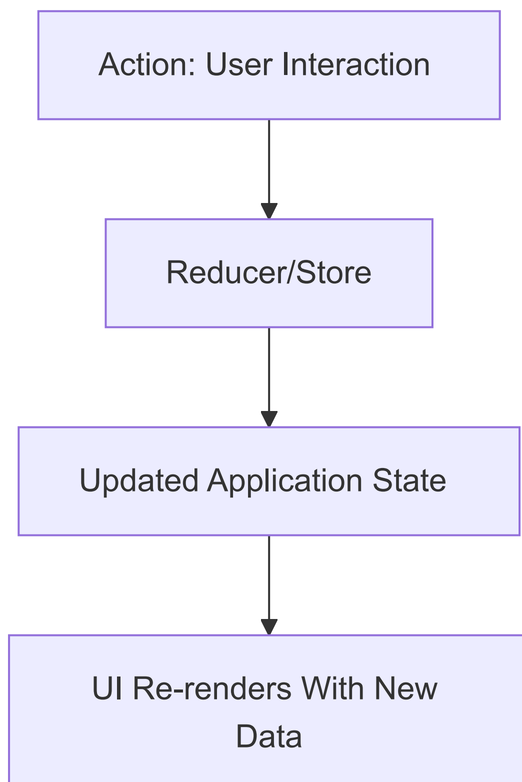
Why State Management Is Needed

Without structured state handling, data can become inconsistent—different components may display conflicting information.

Popular State Management Tools

- **Redux** (React, Vanilla JS)
- **Vuex / Pinia** (Vue)
- **NgRx** (Angular)
- **Svelte Store** (Svelte)

Unidirectional Data Flow



This predictable cycle helps maintain stability and debuggability.

5. How Routing Simulates Navigation in Single-Page Applications

Traditional websites refresh the entire page when navigating between URLs. In contrast, **Single-Page Applications (SPAs)** load a base HTML file once and then dynamically update the content using JavaScript.

What Routing Means in SPAs

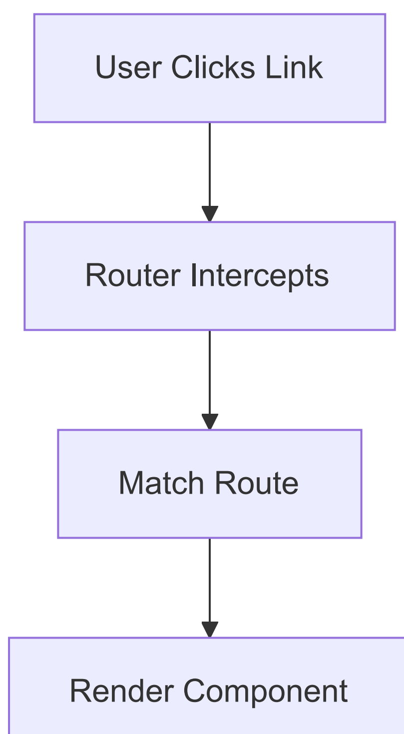
Routing allows the application to *change what is displayed* when the URL changes, **without causing a full page reload**.

Example: `/products`, `/products/42`, `/cart` all display different views using the same loaded page.

How It Works

When the user navigates:

1. The browser URL changes using `history.pushState()`.
2. A **router** matches the URL to a component.
3. The component renders in a defined area of the page.



Popular Routing Libraries:

- **React Router** (React)

- **Vue Router** (Vue)
- **Angular Router** (Angular)
- **SvelteKit Routing** (Svelte)

Benefits of SPA Routing

- Faster transitions
- Less bandwidth usage
- Smoother user experience

Drawbacks

- Requires careful handling for SEO and browser navigation.

---

6. How Browsers and Servers Communicate Through APIs

Modern web applications depend on data retrieved from servers. This communication occurs through **APIs (Application Programming Interfaces)**.

HTTP Requests

APIs are accessed via HTTP methods such as:

Method	Purpose	Example
GET	Retrieve data	Get user list
POST	Create data	Submit a form
PUT/PATCH	Update data	Edit profile
DELETE	Remove data	Delete a product

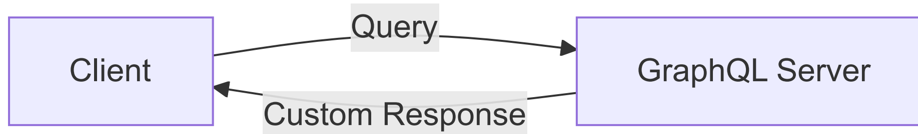
JSON as the Standard Data Format

Servers typically respond with **JSON** because it maps naturally to JavaScript objects.

```
fetch('/api/products')
  .then(res => res.json())
  .then(data => console.log(data));
```

REST vs GraphQL

- **REST** exposes data through URL-based endpoints.
- **GraphQL** allows clients to request exactly the data they need.



---

7. The Role of Build Tools (Webpack, Vite, Babel)

JavaScript in browsers has evolved. Modern code must often be transformed and bundled to ensure compatibility.

Why Build Tools Exist

- To combine multiple JS modules into optimized bundles
- To compile languages like **TypeScript** or **SCSS** to browser-ready formats
- To support older browsers through **transpiling**

Webpack

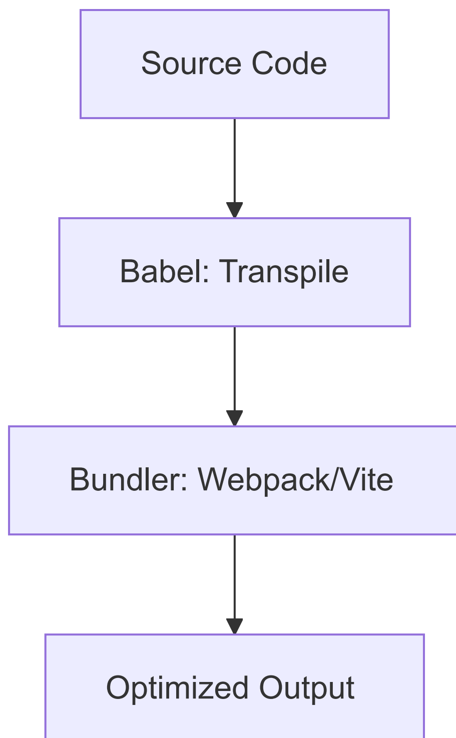
- A highly configurable **module bundler**.
- Allows loaders for CSS, images, TypeScript, etc.

Vite

- Uses **native ES modules** and browser capabilities.
- Provides extremely fast development with **hot module replacement**.

Babel

- A **transpiler** that converts modern JavaScript to versions supported by older browsers.



---

8. Why TypeScript Has Become a Standard

JavaScript is dynamic, which makes it flexible but prone to runtime errors. **TypeScript** introduces **static typing**.

Benefits

- Catches errors during development.
- Improves maintainability in large codebases.
- Provides better editor autocomplete and refactoring tools.

Example:

```
function sum(a: number, b: number): number {
  return a + b;
}
```

Adoption

Most major frameworks (React, Angular, Vue, Svelte) now provide first-class TypeScript support.

---

9. How Testing Ensures Reliability

Testing in frontend development verifies that components and application logic behave as expected. As applications scale, manual testing becomes error-prone, so automated testing is essential.

Types of Frontend Tests

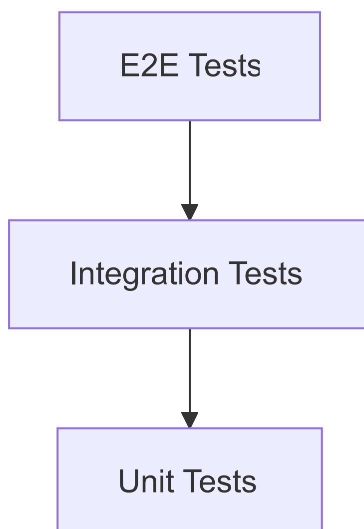
Type	Scope	Tools	Purpose
Unit Tests	Individual functions/components	Jest, Vitest	Ensure small pieces of logic work correctly
Integration Tests	Groups of components working together	Testing Library	Validate component interaction and user flows
End-to-End (E2E)	Full application behavior in browser	Cypress, Playwright	Simulate real user interactions

Example: Unit Testing a Component (React)

```
import { render, screen, fireEvent } from '@testing-library/react';
import Counter from './Counter';

test('increments counter when clicked', () => {
  render(<Counter />);
  fireEvent.click(screen.getByRole('button'));
  expect(screen.getByRole('button')).toHaveTextContent('1');
});
```

Test Pyramid



The base (unit tests) is the largest because they are fast and cheap. E2E tests are fewer but ensure real workflows operate correctly.

---

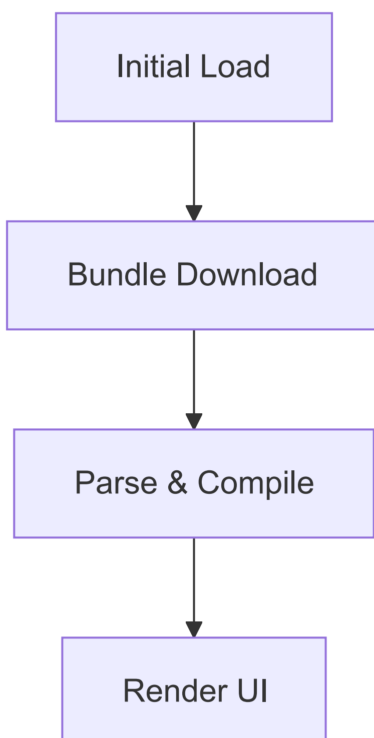
10. How Performance Is Optimized

Modern applications must remain performant even on slower devices or networks.

Key Performance Concepts

- **Minimizing JavaScript bundle size**: Less code → faster load time
- **Lazy loading**: Load components only when needed
- **Memoization**: Avoid recalculating unchanged data
- **DOM virtualization**: Render only visible elements (useful for long lists)

Performance Lifecycle



Optimization strategies must focus on each stage.

Example: Code Splitting (React)

```
const ProductPage = React.lazy(() => import('./ProductPage'));
```

Measurement Tools

- **Lighthouse** (Chrome DevTools)

- **Performance Tab** (browser profiling)
- **Web Vitals** (CLS, FID, LCP metrics)

---

11. How Applications Are Packaged and Deployed

Modern frontend development has evolved far beyond simply writing HTML, CSS, and JavaScript files and uploading them to a web server. Today's frontend applications are often complex single-page applications (SPAs) or multi-page applications (MPAs) built using component-based frameworks, code splitting, static asset optimization, dependencies from npm, and automated build pipelines.

Source Code Organization and Dependencies

A typical frontend application begins in a structured source directory that includes:

- **Components**: UI elements (React `.jsx`, Vue `.vue`, Angular components, etc.)
- **Styles**: CSS/SCSS modules or atomic classes (Tailwind, CSS-in-JS, etc.)
- **JavaScript/TypeScript modules** describing application logic
- **Public assets** such as icons, images, or fonts
- **Configuration files** (e.g., `package.json`, `tsconfig.json`, `.env` files)

Dependencies are managed using **Node.js** package managers such as `npm`, `yarn`, or `pnpm`. These dependencies may include:

- Framework runtimes (`react`, `vue`, `@angular/core`)
- Build system tooling (`vite`, `webpack`, `rollup`, `esbuild`)
- Testing frameworks
- Polyfills or browser compatibility layers

Running:

`npm install`

creates a local dependency tree inside `node_modules`. These dependencies are **not** deployed directly — they are used during the build process to produce optimized output.

The Build Process

The **build step** transforms developer-friendly source code into optimized static assets that browsers can efficiently download and execute.

The **build step** transforms developer-friendly source code into optimized static assets that browsers can efficiently download and execute.

This process is usually handled by a bundler such as:

- **Webpack**
- **Vite** (which uses `esbuild` or `Rollup`)
- **Rollup**
- **Parcel**

Key steps in the build include:

Step	Purpose
Module Bundling	Combines scattered modules into fewer bundled files to reduce network requests.
Tree Shaking	Removes unused code paths to reduce bundle size.
Minification	Strips whitespace, renames identifiers, and compresses code.
Compilation	Transforms TypeScript → JavaScript, JSX → JavaScript, or SCSS → CSS.
Code Splitting	Breaks the codebase into chunks loaded on demand.

The output of the build is a **`dist/`** or **`build/`** directory containing:

```
/dist index.html main.[hash].js vendor.[hash].js styles.[hash].css assets/
```

The hash in filenames ensures **cache busting**—when the code changes, a new filename is generated, forcing browsers to download the updated version instead of using an old cached one.

Environment Configuration

Most frontend apps behave differently depending on environment (development, staging, production). Differences include:

- API base URLs
- Feature flags
- Analytics tokens

Environment-specific configuration is typically handled via:

- `.env` files (e.g., `.env.production`, `.env.staging`)
- Build-time variable injection

Example:

```
VITE_API_URL=https://api.example.com
```

The application reads this **at build time**, meaning the environment must be known during packaging.

CI/CD Pipelines (Continuous Integration / Deployment)

Once code is pushed to a shared repository (GitHub, GitLab, Bitbucket), an automated pipeline typically:

1. Installs dependencies
2. Runs tests and static analysis (ESLint, TypeScript checks)
3. Builds the application
4. Uploads the resulting `dist/` directory to a hosting target

A simplified pipeline flow:

`flowchart LR A[Developer Commit] --> B[CI: Install Dependencies] B --> C[Run Unit/Integration Tests] C --> D[Build Frontend Assets] D --> E[Deploy to Staging or Production]`

Common CI/CD platforms include GitHub Actions, GitLab CI, Jenkins, and CircleCI.

Deployment Targets

Static Hosting on CDNs (Most Common for SPAs)

SPAs are usually deployed to static hosting services:

- Vercel
- Netlify
- AWS S3 + CloudFront
- Google Cloud Storage + CDN
- Azure Static Web Apps
- GitHub Pages

In these setups:

- JavaScript, CSS, and media files are served **as static files**.
- A CDN caches files at edge locations for low-latency access.
- A reverse proxy or rewrite rule ensures `index.html` handles client-side routing.

Example rewrite for SPAs:

`/* → /index.html (200)`

Server-Side Rendering (SSR) Platforms

If using SSR frameworks like Next.js or Nuxt, the deployment involves running a Node.js server or serverless functions to generate pages on-demand.

Caching Strategy and Asset Delivery

Caching is crucial for performance. Typically:

- **Static assets (JS/CSS)**: served with far-future expiration (`Cache-Control: max-age=31536000, immutable`)
- **HTML files**: served with no-cache so updates propagate immediately

This leads to:

File Type	Caching Strategy
`main.[hash].js`, `style.[hash].css`	Heavy caching — safe due to hash versioning
`index.html`	No-cache — must fetch latest file to discover updated asset hashes

Client-Side Bootstrapping

When a user loads the app:

1. The browser requests `index.html`
2. The HTML references the hashed JS/CSS files
3. The framework runtime mounts the application to a DOM root (e.g., `<div id="root"></div>`)
4. The app fetches APIs, initializes state, and renders UI

Example bootstrapping (React):

```
import { createRoot } from 'react-dom/client'; import App from './App'; createRoot(document.getElementById('root')).render(<App />);
```

Rollbacks and Versioning

Because build output is deterministic, deployments can be versioned:

- Each build produces a unique artifact (`dist/`)
- Deployments reference specific versions
- Rollbacks are simply redeploying a previous artifact

On platforms like Vercel or Netlify, this is usually one click.

---

12. Modern Frontend Approaches: SSR, SSG, and Islands Architecture

Modern frontend development is increasingly influenced by performance, SEO, and user experience needs. This has led to architectural approaches that blend server and client rendering.

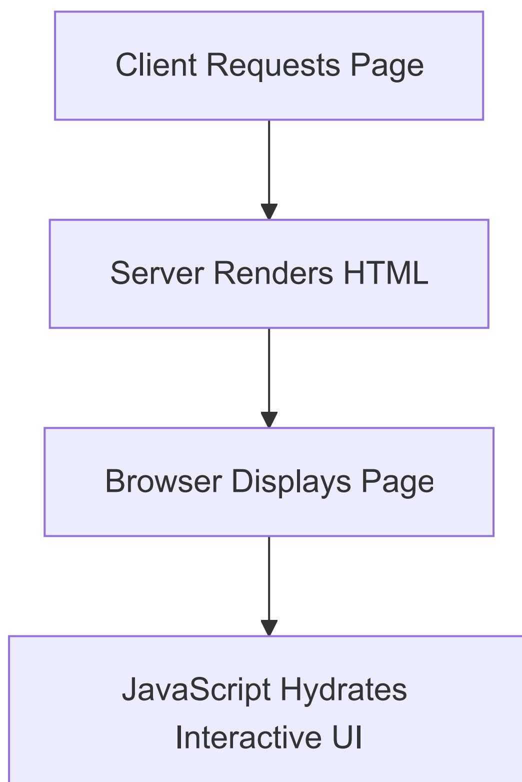
SSR — Server-Side Rendering

In **Server-Side Rendering**, the server generates the HTML for each request.

- Faster initial page load
- Better SEO
- After rendering, JavaScript hydrates the page for interactivity

Frameworks supporting SSR:

- Next.js (React)
- Nuxt (Vue)
- Angular Universal



SSG — Static Site Generation

Pages are generated **at build time** and served from a CDN.

- Extremely fast
- Ideal for mostly-static content (docs, blogs)

Frameworks:

- Gatsby
- Astro
- Next.js static export

Islands Architecture

Only parts of the page are interactive — called **islands**.

- Reduces JavaScript shipped to the browser
- Better performance on low-power devices

Frameworks:

- Astro
- Qwik
- Fresh (Deno)

Comparison Summary

Approach	When HTML is Generated	Best Use Case
CSR (Traditional SPA)	In browser via JS	Highly interactive apps
SSR	On each request	Apps needing SEO or fast first load
SSG	At build time	Content-heavy sites
Islands	Combination of static + minimal hydration	Performance-critical experiences

---

Once development is complete, the application must be packaged and served to users.

Build Output

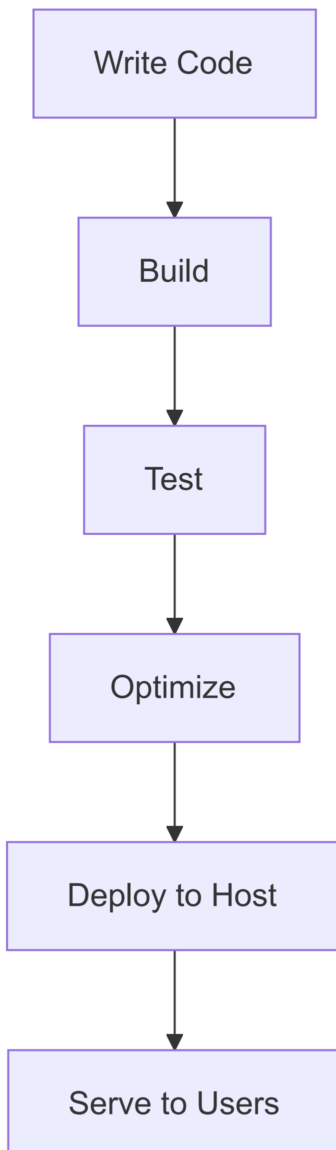
The build process produces:

- **HTML** entry point
- **JavaScript bundles** or ES modules
- **CSS files**
- **Static assets** (images, fonts)

Deployment Models

Platform	Description	Examples
CDN + Static Hosting	Fast global delivery	Netlify, Vercel, Cloudflare Pages
Server-Rendered Hosting	For dynamic content	Next.js, Nuxt, SSR frameworks

Deployment Pipeline



Continuous Deployment (CI/CD)

Modern teams automate deployment using pipelines triggered by Git commits.

Tools: GitHub Actions, GitLab CI, Jenkins.

---

13. Bibliography

MDN Web Docs. (n.d.). *HTML: HyperText Markup Language*. Mozilla. https://developer.mozilla.org/en-US/docs/Web/HTML

MDN Web Docs. (n.d.). *CSS: Cascading Style Sheets*. Mozilla. https://developer.mozilla.org/en-US/docs/Web/CSS

MDN Web Docs. (n.d.). *JavaScript*. Mozilla. https://developer.mozilla.org/en-US/docs/Web/JavaScript

React. (2023). *React Documentation*. Meta. https://react.dev/

Vue.js. (2023). *Vue.js Guide*. https://vuejs.org/guide/

Google. (2023). *Angular Developer Guide*. https://angular.io/docs

Svelte. (2023). *Svelte Tutorial*. https://svelte.dev/tutorial

Redux. (2023). *Redux: A Predictable State Container for JS Apps*. https://redux.js.org/

Next.js. (2023). *Next.js Documentation*. Vercel. https://nextjs.org/docs

Astro. (2023). *Astro Documentation*. https://docs.astro.build

Google Chrome Developers. (2023). *Web Vitals*. https://web.dev/vitals/

ECMA International. (2023). *ECMAScript Language Specification*. https://tc39.es/ecma262/