Guides

Front-End Development

From the static-document web to modern React and Next.js — the journey from imperative DOM scripting to declarative UI frameworks.

A Brief History

The Static Web (1989–1995)

Tim Berners-Lee proposed the World Wide Web at CERN in 1989 and put up the first website on August 6, 1991. The earliest web was a network of static documents — HTML files served over HTTP, linked together with hyperlinks. There was no interactivity, no styling beyond the browser's defaults, and no scripting. A page looked the same to every visitor and only changed when an author re-saved the file.

CSS arrived in 1996 as a way to separate presentation from content. By 2000, the three-language stack — **HTML for structure, CSS for style, JavaScript for behavior** — was set, and it has not changed since. Every modern framework, including the ones you'll learn this quarter, ultimately compiles down to those three.

JavaScript Makes Pages Interactive (1995–2005)

JavaScript landed in Netscape Navigator in 1995. (See the TypeScript guide overview for the language's full origin story.) For nearly a decade, JavaScript on the web meant form validation, image rollovers, and the occasional drop-down menu. Pages were still server-rendered — every meaningful interaction triggered a full page reload.

The hinge moment was `XMLHttpRequest` (XHR), introduced by Microsoft in IE5 in 1999. XHR let a script send an HTTP request *without* navigating the page. By 2005, Google was using XHR in Gmail and Google Maps, and Jesse James Garrett's essay coined the term **AJAX** (Asynchronous JavaScript and XML). Pages could now update in place — fetch new data, render it, update the DOM — without a server round-trip. This is the moment the web became an application platform, not just a document platform.

The jQuery Era (2006–2013)

By 2006, browsers had drifted apart in incompatible ways. The same JavaScript code would behave differently in IE6, Firefox, and Safari. **jQuery**, released by John Resig in 2006, became the industry-standard answer: a thin layer that hid the browser inconsistencies and offered a clean, chainable API for DOM manipulation, AJAX, and animation. At its peak, jQuery was loaded by the vast majority of websites on the public web.

jQuery solved real problems, but the problems it solved were largely the symptoms of writing imperative DOM code at scale. As applications grew — Gmail, Facebook, Google Docs — engineers found themselves writing thousands of lines of "find this element, mutate that property, hope the state stays consistent." There had to be a better model.

The React Era (2013–Present)

In May 2013, Facebook open-sourced **React** at JSConf US. React inverted the problem: instead of telling the browser *how* to update the DOM step by step, you describe *what* the UI should look like for the current state, and React figures out the DOM mutations. The mental model is `UI = f(state)`.

React did not win immediately. AngularJS (Google, 2010) and Ember (2011) were established alternatives, and Vue.js (2014) emerged as a lightweight competitor. By 2018, React had the largest ecosystem and mindshare; in 2026 it is the default choice for new web applications and the front-end framework most students will encounter on their first job.

Next.js and the Framework Era (2016–Present)

React is "just the view layer." It does not handle routing, server rendering, build tooling, or data fetching. Around React, the ecosystem has built **meta-frameworks** — opinionated wrappers that decide all of those concerns for you. **Next.js**, released by Vercel (then called Zeit) in October 2016, is now the dominant React meta-framework. Its **App Router**, introduced in Next.js 13 (2022), added **React Server Components** and made data fetching feel like server-rendered code that happens to use React.

This is where TCSS 460 plants its flag. By the end of the quarter, you will have built a multi-page Next.js app that signs users in via OAuth2, calls your partner team's REST API with a bearer token, and deploys to Vercel or Render with one `git push`.

Why React and Next.js for TCSS 460?

You already know how to build a back-end API. The front-end half of the course completes the picture: you need to build a user-facing application that consumes that API, holds an authenticated session, and ships to a public URL.

The same problems are still being solved as in the jQuery era — the mental model just changed:

Concept	Vanilla DOM / jQuery	React / Next.js
Updating the UI	Imperative (`el.textContent = "..."`)	Declarative (`{value}` in JSX)
Component reuse	Copy/paste HTML	Function components
State management	Ad-hoc, scattered across globals	`useState`, lifted state, context
Routing	Server-side or hash-fragment hacks	File-based routing (Next.js App Router)
Data fetching	Manual `fetch` + DOM updates	Same `fetch` primitive, framework conventions
Authentication	Hand-rolled session cookies	NextAuth (Auth.js)
Deployment	FTP a folder to a static host	`git push` → Vercel / Render

The bottom row is worth noticing: the platform primitives have not changed. The browser still parses HTML, applies CSS, runs JavaScript, and exposes the DOM. `fetch` is still the only way the browser sends an HTTP request. React and Next.js are abstractions *on top of* the same primitives that have powered the web since 1995. Learning the substrate first — what the Fundamentals guides do — pays dividends every time the abstraction leaks.

Choosing React and Next.js for this course means:

- **Transferable skills.** Every front-end internship and entry-level role you'll apply for assumes some React knowledge.
- **One language across the stack.** Your Express API and your Next.js front end are both written in TypeScript — you carry one mental model across the wire.
- **Real OAuth2 integration.** NextAuth combined with the course's auth-squared provider lets you do federated sign-in the way industry actually does it, not a toy username/password form.
- **Production deploys for free.** Vercel and Render both have free tiers that handle everything you'll do in this course.

Sources and Further Reading ›

Guides

These guides walk you from a static page to a deployed, authenticated Next.js application. Read them in order during Weeks 6 and 7.

Fundamentals

The browser substrate every framework sits on top of. Start here, even if you've used React before.

- HTML, CSS & the DOM — Document structure, styling, and DOM manipulation
- Consuming a Web API from the Browser — `fetch`, async/await, error handling, CORS, bearer tokens
- Accessibility — Semantic HTML, keyboard navigation, ARIA, contrast, screen-reader testing

Modern

React and the Next.js framework around it.

- React Fundamentals — Components, JSX, props, state, hooks, and side effects
- Next.js — File-based routing, server vs client components, the App Router

Deployment

Take your Next.js app live on a public URL.

- Deploying a Next.js App — Vercel and Render, env vars, custom domains, troubleshooting