Next.js Course

Flavio Copes

Updated August 3, 2026

Contents

Learn the Next.js App Router by building routes, Server and Client Components, data loading, forms, Server Actions, Route Handlers, and a production build.

What you'll learn: Build a production-shaped App Router application with nested and dynamic routes, server data, a validated mutation, useful failure states, and a small HTTP endpoint.

App Router foundations

Create a current project, read its structure, and place Server and Client Component boundaries.

What Next.js adds to React

Understand the framework features around React and why the App Router is the right starting point for a new Next.js project.

React builds interfaces from components. Next.js adds routing, server rendering, data access, request handlers, asset optimization, and a production build system around those components.

This course uses the App Router, the current file-system router built around Server Components, Suspense, and Server Functions. You do not need Vercel to run Next.js, and you do not need to replace every server with Next.js. Use it when one project benefits from a React interface and server capabilities that work together.

It helps to picture three outputs. The server sends HTML so the first screen can appear, a React Server Component payload describes the server-rendered component tree, and JavaScript hydrates only the Client Components that need interaction. These are related, but they are not the same artifact.

That distinction explains many design decisions. Database access belongs in server code because it needs credentials. A click handler belongs in client code because it runs in the browser. A route may combine both without turning the whole page into browser JavaScript.

For every feature, ask three questions: where does this code execute, what data crosses the boundary, and what does the user receive if JavaScript is slow or unavailable? Those answers are more useful than calling an entire page "server rendered" or "client rendered."

Sketch one interface you could build in this course. Give it at least two pages, one dynamic page, and one form that changes data. Mark each piece as build-time, request-time server work, or browser work, then explain one boundary choice.

Create an App Router project

Generate a current TypeScript project with the recommended defaults, run it locally, and identify the development and production commands.

Start with the official generator. Its defaults change as the framework evolves, so using the current command is safer than copying an old starter repository.

Current Next.js requires Node.js 20.9 or newer. Check `node --version` before generating the project so a framework error is not mistaken for an application error.

The recommended defaults enable TypeScript, Tailwind CSS, ESLint, the App Router, Turbopack, and the `@/*` import alias. Record the choices and installed Next.js version in the project README. Commands and defaults from an older tutorial may describe a different caching or rendering model.

The development server optimizes for fast feedback and recompiles routes on demand. `next build` analyzes and prepares the production application; `next start` serves that output. A page working in development is therefore useful feedback, not proof that the production build succeeds.

```
npx create-next-app@latest field-notes --yes
cd field-notes
npm run dev
```

```
# later:
npm run build
npm start
```

Create the project, open http://localhost:3000, and inspect the terminal request log. Stop the development server, run a production build, and serve it once before changing any code. Save the version and generator choices so later framework-specific behavior can be traced.

Read the project structure

Locate routes, public files, global styles, configuration, and dependencies without treating every generated file as magic.

The important folders are small enough to understand. `app` contains routes and route-specific files, while `public` contains files served from the site root.

`app/layout.tsx` is the required root layout and `app/page.tsx` is the home page. `app/globals.css` holds global styles. `next.config.ts` changes framework behavior, `tsconfig.json` configures TypeScript and aliases, and `package.json` records scripts and dependencies. Files outside `app` do not become routes merely because they contain components.

Inside `app`, ordinary files are also safe to colocate. A folder becomes public only through a special file such as `page.tsx` or `route.ts`. Files including `layout.tsx`, `loading.tsx`, `error.tsx`, and `not-found.tsx` add behavior to a route segment instead of creating separate URLs.

Keep route-specific components and data functions near the route until another route genuinely shares them. Put broadly shared application code in folders such as `components` or `lib`. This makes ownership visible without inventing one global folder for every kind of file.

Files in `public` bypass the component and module pipelines and are served from `/`. The `.next` directory is generated output: inspect it when debugging, but never treat it as source or edit it by hand.

Replace the generated home page with a heading and short introduction. Add a route-local component beside it and a plain file to `public`; confirm only the special page file creates a URL and the public file is available from a root-relative path.

Separate Server and Client Components

Keep components on the server by default and introduce a focused client boundary only when an interface needs browser interactivity.

Pages and layouts in the App Router are Server Components by default. They can read server-side data and secrets without shipping their component code to the browser.

Add `"use client"` at the top of a file when it needs state, event handlers, effects, context, or browser-only APIs. That directive creates a client boundary for the file and its imports. Keep the boundary small: a server page can render a focused interactive child and pass serializable data into it.

On the first request, Next.js uses the server result to produce HTML for a fast initial display and a React Server Component payload for reconciliation. Client Component JavaScript then hydrates the interactive pieces. On later client navigation, the router can use the server payload without reloading the whole document.

The directive marks a module boundary, not merely one component. Modules imported by that file join the client graph, so putting it high in the tree can ship data-formatting libraries and otherwise static UI to the browser. Pass the smallest serializable props across the boundary; functions, database handles, and arbitrary class instances are not ordinary client props.

```
// app/counter.tsx
'use client'

import { useState } from 'react'

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

Create the Counter component and render it from the server home page. Inspect the browser's JavaScript requests, then move the boundary to the page temporarily and compare the client graph. Remove `"use client"` from the counter once to read the state-hook error before restoring it.

Check your understanding: app router foundations

Check the role of Next.js, the App Router, the project structure, and the boundary between Server and Client Components.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Routing and navigation

Build pages, layouts, nested and dynamic routes, and semantic navigation.

Create pages and layouts

Turn folders into URL segments and use layouts for interface that should persist while child routes change.

A folder inside `app` becomes a public route only when it contains a special route file such as `page.tsx`. A `layout.tsx` wraps pages below its segment.

Create `app/notes/page.tsx` for `/notes`. Add `app/notes/layout.tsx` when all note routes share navigation or structure. Layouts preserve state across navigation and receive the active child as `children`; ordinary reusable components do not need to be layouts.

The root layout is required and owns the document's `<html>` and `<body>` elements. Nested layouts should express persistent route structure: navigation, a workspace shell, or a sidebar shared by descendants. Do not use a layout merely to avoid importing a component.

Persistence is visible during client navigation. A nested layout is not remounted as its child page changes, so local client state inside that layout can survive. A full reload creates a new document and resets it. Design with that lifecycle in mind rather than assuming every navigation starts the tree again.

```
// app/notes/layout.tsx
export default function NotesLayout({ children }: { children: React.ReactNode }) {
  return (
    <section>
      <h1>Field notes</h1>
      {children}
    </section>
  )
}
```

Add `/notes` and a notes layout. Put a tiny client counter in the layout, navigate between two child pages, and compare that with a full reload. The result should make the layout lifecycle concrete.

Nest and organize routes

Build deeper URLs, use route groups without changing those URLs, and keep private implementation folders out of the route tree.

Nested folders map naturally to nested paths: `app/notes/new/page.tsx` produces `/notes/new`. This makes route structure visible in the filesystem.

A folder in parentheses, such as `app/(marketing)`, is a route group and does not add a URL segment. A private folder prefixed with `_` is excluded from routing. Use these conventions to organize layouts and colocated code, not to disguise an unnecessarily complex URL structure.

Route groups are useful when two sections need different layouts without adding organizational words to their URLs. Because the group name disappears, `(marketing)/about/page.tsx` and `(company)/about/page.tsx` still compete for `/about`; Next.js cannot publish both. Multiple root layouts also cause a full page load when navigation crosses between them.

Private folders are an explicit signal that a subtree is implementation detail. Ordinary colocated files are already non-routable, so use `_components` or `_lib` when the signal improves navigation—not as compulsory ceremony. Shared domain rules should still live outside a route when several route trees use them.

Create `/about` inside a `(marketing)` group and `/notes/new` inside the notes route. Add one private `_components` folder, then predict every public URL from the tree before opening the browser.

Build dynamic routes

Represent resource identifiers with dynamic segments and read the resolved parameters in an App Router page.

You do not create one file for every note. A dynamic segment such as `[slug]` captures that part of the URL and lets one page render many resources.

Create `app/notes/[slug]/page.tsx`. In current Next.js, `params` is asynchronous, so await it before reading the value. Validate the slug before using it in a query; a value coming from the URL is untrusted input.

Dynamic does not necessarily mean request-only. If the set of slugs is known at build time, `generateStaticParams` can return them for prerendering. Other values may still be rendered on demand unless the route configuration deliberately rejects them. Choose that behavior from the data lifecycle, not from the bracket syntax alone.

```
export default async function NotePage({
  params,
}: {
  params: Promise<{ slug: string }>
}) {
  const { slug } = await params
  return <p>Reading {slug}</p>
}
```

Open `/notes/first-note` and `/notes/another-note`. Decode and validate the slug before it reaches the data layer, return `notFound()` for a valid-looking slug with no record, and use `generateStaticParams` for two known notes. Compare the production build output with the on-demand case.

Link and navigate

Use Link for ordinary navigation, redirect on the server, and reserve useRouter for interactions that cannot be expressed as links.

Navigation is part of the interface. Use `next/link` for links so Next.js can prefetch and perform client-side transitions while keeping real anchor semantics.

Use a normal `<a>` for an external site or download. Call `redirect()` during server work when rendering should continue at another URL. Use `useRouter` from `next/navigation` in a Client Component for programmatic transitions after an interaction. A clickable destination should usually remain a Link, not a button with `router.push()`.

In production, Next.js may prefetch linked routes when they enter the viewport, then reuse prefetched data for a faster transition. Treat that as an optimization. Every destination must still work when loaded directly, refreshed, bookmarked, or opened in a new tab. Never put a mutation or another side effect in page rendering merely because you expect users to arrive through a click.

Use the router only when navigation is the result of code, such as completing a multi-step interaction. Preserve normal link behavior for destinations so keyboard navigation, modified clicks, copying the URL, and browser history all continue to work.

```
import Link from 'next/link'

export function NoteLink({ slug }: { slug: string }) {
  return <Link href={`/notes/${slug}`}>Read note</Link>
}
```

Add Link navigation for Home, Notes, and New note. Verify keyboard and modified-click behavior, then use the Network panel to compare a client transition with a direct request. Both paths must render the same correct page.

Check your understanding: routing and navigation

Check pages, layouts, nested and dynamic segments, route groups, links, and programmatic navigation.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Data and rendering

Load server data, stream UI, handle failures, cache deliberately, and add metadata.

Fetch data on the server

Load data directly in an asynchronous Server Component and keep credentials and database access out of the browser bundle.

A Server Component can await `fetch()`, an ORM, a database driver, or filesystem work directly. You do not need an effect just to load the initial page data.

Check HTTP responses before parsing them and return a useful empty or error state. Server-side code may read non-public environment values, but rendered output and props passed to Client Components still reach the user. Never serialize a secret into the interface.

Call the underlying data function directly when it lives in the same application. Fetching your own Route Handler adds an HTTP hop, duplicates error translation, and makes the server depend on its own public URL. Keep one domain function that a page, action, and Route Handler can each call after applying their own boundary checks.

Current Next.js does not make every `fetch` persistently cached. Decide whether the result must be fresh, request-memoized, or reusable across requests, then express that choice with the APIs supported by the installed version. Authentication and per-user data are usually request-specific; a public catalog may tolerate deliberate caching.

Shape data before rendering it. A query returning whole user records can leak fields even if the JSX only seems to use a name later. Authorization should select the records this user may see, and projection should select the fields the interface needs.

```
export default async function NotesPage() {
  const response = await fetch('https://api.example.com/notes')
  if (!response.ok) throw new Error('Could not load notes')
  const notes = await response.json()
  return <pre>{JSON.stringify(notes, null, 2)}</pre>
}
```

Load data from a small public endpoint or local data function. Handle non-2xx, empty, and malformed responses, and document the freshness decision. Log on the server and confirm neither the log nor an unused private field reaches the browser.

Stream useful loading UI

Use loading.tsx and focused Suspense boundaries so slow data does not leave navigation without feedback.

Server rendering can stream route output. A `loading.tsx` file gives the route segment an immediate fallback while its page is prepared.

Use a route-level loading file for a broad page skeleton. Use `<Suspense>` closer to a slow component when the rest of the page can render first. Make the fallback resemble the space it will replace so the page does not jump when content arrives.

The route fallback is prefetched during navigation and wraps the page below its shared layout. This lets navigation feel immediate while the server finishes the segment. On a direct request, streaming still depends on the server and hosting path delivering chunks rather than buffering the entire response.

Prefer the narrowest meaningful boundary. If only the recent-notes list is slow, keeping the heading and create button outside Suspense leaves useful work available. Several tiny boundaries can create visual noise and a waterfall, so split by user-visible units rather than by every asynchronous function.

```
// app/notes/loading.tsx
export default function Loading() {
  return <p aria-live="polite">Loading notes…</p>
}
```

Add an intentional delay, create a route loading file, and navigate through a Link. Then replace the broad fallback with Suspense around only the slow list. Compare client navigation and a direct load, including the layout space reserved by each fallback.

Handle errors and missing data

Treat validation failures and missing resources as expected outcomes while containing unexpected exceptions with route boundaries.

A missing note is not the same as a crashed database client. Expected outcomes should be represented deliberately instead of thrown as generic exceptions.

Call `notFound()` for an absent resource and provide `not-found.tsx`. Return structured validation results from actions. Add `error.tsx` as a Client Component for uncaught exceptions within a route segment; it receives a reset function for retrying. Log the underlying server error without exposing private details to the visitor.

An error boundary handles failures in the segment below it, not failures thrown by the layout at that same level. Move the boundary up when shared layout work can fail, or move risky work into a child segment. A global error boundary is the last resort for failures at the root.

Calling `reset()` asks React to render the segment again. It can recover from a temporary failure, but it cannot repair invalid configuration or a deterministic bug. Give the visitor another route out and preserve a server-side error identifier so operators can connect a friendly message to the real failure. In production, Server Component error details are intentionally sanitized.

Make one dynamic slug call `notFound()`. Add a route error boundary, trigger a separate temporary exception and a permanent one, and observe what reset can and cannot fix. Check that the production response reveals no stack, query, or secret.

Cache, revalidate, and describe a page

Make caching an explicit decision, refresh stale results after mutations, and add route metadata without relying on obsolete defaults.

Do not assume every server fetch is automatically cached. Current Next.js renders uncached work dynamically unless you deliberately opt data or output into caching.

With Cache Components enabled, use `"use cache"` for reusable cached work, `cacheLife` to describe its lifetime, and `cacheTag` to give related entries a domain name. The values passed into a cached function form part of its cache key, so accept the note ID instead of closing over mutable request

state.

After a mutation, choose invalidation by meaning. `updateTag` gives the mutating user read-your-own-writes behavior. `revalidateTag` marks tagged data for refresh according to its profile, while `revalidatePath` invalidates a route so its next visit can regenerate. Invalidating every page hides unclear ownership and throws away useful cache work.

Without Cache Components, follow the caching model documented for the exact installed Next.js version. Do not mix examples from both systems into one app. Record the version and chosen model beside the code so a future upgrade can review the assumptions.

Use the Metadata API for titles and descriptions; dynamic pages can export `generateMetadata` from server code. Metadata has a freshness contract too: if it reads the same note as the page, reuse the data function and invalidation rules rather than allowing the tab title to lag behind the content.

```
export const metadata = {
  title: 'Field Notes',
  description: 'Short observations from the field',
}

export default function Page() {
  return <h1>Field Notes</h1>
}
```

Add static list metadata and dynamic note metadata, then inspect the document head after a rename. Document whether the project uses Cache Components, tag cached note data, mutate it, and verify exactly which visit observes the new value.

Check your understanding: data and rendering

Check server-side data access, streaming, expected and unexpected errors, caching, revalidation, and metadata.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Actions and forms

Validate FormData, mutate on the server, revalidate views, and show useful pending states.

Create a Server Action

Connect a form to an asynchronous server function without writing a separate client-side request handler.

React lets a form call a Server Function through its `action` prop. In a mutation context that function is commonly called a Server Action.

Mark the async function with `"use server"`, inline in a Server Component or at the top of a dedicated actions file. The browser sends a POST request behind the scenes. Keep authorization and validation inside the action because a user can invoke the server endpoint without using your

rendered form.

With a Server Component form, the browser can submit before client JavaScript loads, so the basic mutation can progressively enhance native HTML. A form rendered inside a Client Component may queue submission until hydration. Do not make correctness depend on that queue or on a disabled button.

Treat every action as a public mutation endpoint. Hidden inputs and IDs bound into an action are user-controlled intent, not authorization. Read the current user on the server, check access to the specific record, validate the full payload, and return only safe serializable state.

```
async function createNote(formData: FormData) {
  'use server'
  const title = formData.get('title')
  // validate, authorize, then write
}

export default function Page() {
  return <form action={createNote}><input name="title" /><button>Save</button></form>
}
```

Create the form and action. Log the received value on the server, submit with JavaScript disabled, then alter a note ID in the request. The server should reject the forged resource even though the visible form never offered it.

Validate FormData

Convert unknown form values into a trusted shape and return field-level errors instead of assuming TypeScript checked runtime input.

`formData.get()` can return a string, a File, or null. Type annotations do not validate any of those runtime values.

Normalize the input, validate required fields and bounds, and return a serializable result for expected errors. A schema library is useful in a larger form, but the important boundary is the same: do not perform the write until server-side validation succeeds.

Authentication answers who made the request; validation answers whether the submitted shape is acceptable; authorization answers whether that person may perform this operation. Run all three before the write. Client constraints improve feedback but can be removed or bypassed.

```
const rawTitle = formData.get('title')
if (typeof rawTitle !== 'string' || rawTitle.trim().length < 3) {
  return { errors: { title: 'Use at least 3 characters' } }
}
const title = rawTitle.trim()
```

Do not echo raw values or internal exception messages in the returned state. Preserve safe user-entered values separately when the form should be repopulated, and use a general message for an unexpected storage failure.

Reject missing, `File`, whitespace-only, too-short, and very long titles by constructing `FormData` directly in a small test. Confirm each expected problem returns a field message and performs no write.

Mutate and revalidate

Write data once, invalidate the affected view, and redirect only after the mutation has completed successfully.

A successful mutation can leave cached or previously rendered data stale. The action should identify which view or cache entry is now invalid.

Perform authorization, validation, and the write first. Then call the revalidation API that matches your caching model, such as `revalidatePath('/notes')`, and optionally redirect to the new resource. Do not redirect from inside a catch block that accidentally treats the framework redirect as an error.

Invalidate only after the write commits. Doing it before a failed write can regenerate the old data and leave that stale result cached again. When several writes form one operation, use a transaction so the cache never advertises a half-finished state.

`redirect()` throws a framework control-flow signal and ends the action. Put it after the error-handling region. Also design for retries: a double click, network retry, or resubmitted browser request can invoke the action more than once. A unique constraint or idempotency key protects the data; a pending button alone does not.

```
'use server'
import { revalidatePath } from 'next/cache'
import { redirect } from 'next/navigation'

export async function createNote(formData: FormData) {
  const note = await saveValidatedNote(formData)
  revalidatePath('/notes')
  redirect(`/notes/${note.slug}`)
}
```

Connect the action to persistence, revalidate the smallest relevant path or tag, and redirect to the created note. Simulate a failed write and a repeated request; neither should create a false success or duplicate record.

Show pending and result states

Use React form hooks to prevent duplicate work and render server validation messages without rebuilding the submission flow by hand.

A form should explain that work is in progress and show expected server results near the relevant controls.

`useFormStatus` reads the status of a parent form and is useful in a nested submit button. `useActionState` keeps the serializable result returned by an action. These hooks require Client Components, but the action itself remains server code. Keep native labels, names, and constraints so the form stays understandable before enhancement.

The status hook must run in a component nested inside the form it observes. It does not describe arbitrary requests elsewhere on the page. Use the pending state to prevent confusing repeat interaction and to announce progress, while keeping navigation and recovery actions available.

Connect field messages to inputs with stable IDs and `aria-describedby`, and set `aria-invalid` when a field failed. Keep an unexpected server failure visible until the user retries; a disappearing

toast is poor evidence that data was not saved. Success should move focus only when the next task genuinely requires it.

```
'use client'
import { useFormStatus } from 'react-dom'

export function SubmitButton() {
  const { pending } = useFormStatus()
  return <button disabled={pending}>{pending ? "Saving…" : "Save"}</button>
}
```

Add a delayed action, disable only its submit button, and render an associated title error plus a persistent general failure. Submit twice quickly and remember that the server still needs idempotency even when the UI appears to block the second click.

Check your understanding: actions and forms

Check Server Functions, FormData, validation, mutations, revalidation, pending states, and progressive enhancement.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Build a production-shaped app

Add an HTTP endpoint, protect configuration, optimize assets, and verify the production build.

Add a Route Handler

Expose an HTTP endpoint with the Web Request and Response APIs and choose it only when the application needs a real protocol boundary.

A `route.ts` file inside `app` handles HTTP methods such as GET and POST. It is the App Router equivalent of an API route.

Use Route Handlers for webhooks, public APIs, downloads, feeds, or clients that cannot call a Server Action. Do not make a Server Component call your own Route Handler just to reach the same server-side data function; call the shared function directly. A route file cannot coexist with a page file at the same segment.

Each exported method receives the standard Web `Request` object and returns a standard `Response`. This makes the protocol boundary visible: parse and authenticate the request, call trusted application code, then translate the result into an HTTP status, headers, and body. Unsupported methods receive `405 Method Not Allowed`.

```
// app/api/health/route.ts
export async function GET() {
  return Response.json(
    { ok: true, checkedAt: new Date().toISOString() },
    { headers: { 'Cache-Control': 'no-store' } },
  )
}
```

A GET handler is not automatically a promise of static or cached data. Its behavior depends on the APIs it uses and, with Cache Components, whether you deliberately cache the underlying work. Make freshness part of the endpoint contract.

For a webhook, verify the signature against the raw body before parsing or mutating anything. For an authenticated API, authorize the specific resource rather than merely checking for a session. Return stable public error codes, not stacks or database messages.

Add the health route and inspect its status, content type, and cache header with curl. Send POST too and confirm it receives 405. The handler should call shared server code directly instead of making a loopback request.

Protect environment values

Keep server configuration outside source control and understand exactly when a value becomes part of browser-visible code.

Next.js loads environment files for local work. Values without the `NEXT_PUBLIC_` prefix remain server-only when used in server code.

A public prefix deliberately inlines a value into browser code at build time, so it must never hold a secret. The public value is frozen when the app is built: promoting the same artifact to another environment does not replace it at runtime.

Keep `.env.local` ignored, provide a documented `.env.example` with names but no values, and configure real values in the deployment platform. Validate required configuration near the server entry point rather than failing deep inside a request.

```
// lib/server/config.ts
import 'server-only'

const notesApiToken = process.env.NOTES_API_TOKEN
if (!notesApiToken) throw new Error('NOTES_API_TOKEN is required')

export const serverConfig = { notesApiToken }
```

`server-only` turns an accidental Client Component import into a build error. It is a guardrail, not a vault. A value still leaks if server code renders it into HTML, passes it as a client prop, includes it in an error, or logs it somewhere other people can read.

Add `NOTES_API_TOKEN` through the guarded config module and its empty name to `.env.example`. Deliberately import the module from a Client Component to observe the failure, then remove the import. Inspect Git, HTML, client props, built JavaScript, and production logs for the token value.

Use optimized images and fonts

Render appropriately sized images and self-hosted font files through the framework APIs without forgetting semantics or layout stability.

`next/image` helps size, lazy-load, and serve responsive images. `next/font` loads font files with predictable layout and no browser request to a third-party font provider.

Importing a local image gives Next.js its intrinsic dimensions. A remote image needs explicit `width` and `height`, or `fill` inside a positioned container with a real size. Restrict remote sources with `remotePatterns`; an unrestricted image proxy can fetch content you never intended to serve.

```
<Image
  src={noteCover}
  alt="Handwritten notes beside an open laptop"
  sizes="(max-width: 768px) 100vw, 640px"
/>
```

`sizes` tells the browser how wide the image will render so it can choose a suitable candidate. Without an accurate value, a responsive image can still download far more pixels than it needs. Reserve preload or high-priority loading for the image that materially affects the first viewport.

The APIs do not choose useful content for you. Explain the image's purpose in alt text, or use `alt=""` when it is decorative, and preserve dimensions to avoid layout shift. With `next/font`, expose the returned class or CSS variable near the root layout and request only the families, subsets, styles, and weights the interface uses.

Add one meaningful image and one font. At narrow and wide sizes, inspect dimensions, alt text, the selected `srcset` candidate, layout shift, and font requests. Remove an unused weight and verify the network evidence changes.

Build and review Field Notes

Complete a small App Router application and verify routes, mutations, server boundaries, errors, accessibility, and its production build.

Finish the project by connecting the pieces: a notes list, dynamic note pages, a new-note form, useful loading and error states, and a health endpoint.

Review it as a system, not a set of screenshots. Write a route map that names what runs at build time, on the request-time server, and in the browser. Confirm Client Component boundaries contain only the interaction that needs them and receive serializable props.

Test direct URL loads and client navigation because they exercise different paths. Request a valid note, an unknown note, invalid parameters, and a deliberately failing data source. Slow the data request and check that the loading boundary appears without replacing more of the page than necessary.

Exercise mutations separately. Submit valid and invalid forms, repeat a request, and confirm authorization and validation run on the server. Check that committed data becomes visible through the chosen invalidation strategy and that a failed write does not redirect or claim success. A disabled button improves feedback; idempotent server behavior protects data.

Finally, build with a missing environment value and inspect HTML and browser JavaScript. Check image candidates, font downloads, keyboard navigation, focus after errors, response statuses, and server logs. Development success is not production evidence: rendering, caching, and error output can differ.

```
npm run lint
npm run build
npm start
```

Complete the application and write a short README with its routes, runtime boundaries, configuration, cache decisions, and verification commands. Ask another person to create and open a note using only the interface, then record every place their path differed from yours. Deployment remains a separate concern covered by the Vercel course.

Check your understanding: build a production-shaped app

Check Route Handlers, environment variables, public data boundaries, optimized assets, production builds, and final review habits.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/nextjs/.