# Learn Next.js

Welcome to the Next.js App Router course! In this free interactive course, you'll learn the main features of Next.js by building a full-stack web application.

## [What we'll be building](https://nextjs.org/learn/dashboard-app#what-well-be-building)

![Screenshots of the dashboard project showing desktop and mobile versions.](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fdashboard.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

For this course, we'll be building a simplified version of the financial dashboard that has:

- A public home page.

- A login page.

- Dashboard pages that are protected by authentication.

- The ability for users to add, edit, and delete invoices.

The dashboard will also have an accompanying database, which you'll set up in [a later chapter](https://nextjs.org/learn/dashboard-app/setting-up-your-database).

By the end of the course, you'll have the essential skills needed to start building full-stack Next.js applications.

## [Overview](https://nextjs.org/learn/dashboard-app#overview)

Here's an overview of features you'll learn about in this course:

- \*\*Styling\*\*: The different ways to style your application in Next.js.

- \*\*Optimizations\*\*: How to optimize images, links, and fonts.

- \*\*Routing\*\*: How to create nested layouts and pages using file-system routing.

- \*\*Data Fetching\*\*: How to set up a database on Vercel, and best practices for fetching and streaming.

- \*\*Search and Pagination\*\*: How to implement search and pagination using URL Search Params.

- \*\*Mutating Data:\*\* How to mutate data using React Server Actions, and revalidate the Next.js cache.

- \*\*Error Handling:\*\* How to handle general and `404` not found errors.

- \*\*Form Validation and Accessibility:\*\* How to do server-side form validation and tips for improving accessibility.

- \*\*Authentication\*\*: How to add authentication to your application using [`NextAuth.js`](https://next-auth.js.org/) and Middleware.

- \*\*Metadata\*\*: How to add metadata and prepare your application for social sharing.

## [Prerequisite knowledge](https://nextjs.org/learn/dashboard-app#prerequisite-knowledge)

This course assumes you have a basic understanding of React and JavaScript. If you're new to React, we recommend going through our [React Foundations](https://nextjs.org/learn/react-foundations) course first to learn the fundamentals of React, such as components, props, state, and hooks, and newer features like Server Components and Suspense.

## [System requirements](https://nextjs.org/learn/dashboard-app#system-requirements)

Before you start this course, make sure your system meets the following requirements:

- Node.js 18.17.0 or later installed. [Download here](https://nodejs.org/en).

- Operating systems: macOS, Windows (including WSL), or Linux.

In addition, you'll also need a [GitHub Account](https://github.com/join/) and a [Vercel Account](https://vercel.com/signup).

## [Join the conversation](https://nextjs.org/learn/dashboard-app#join-the-conversation)

If you have questions about this course or would like to provide feedback, you can ask our community on [Discord](https://discord.com/invite/Q3AsD4efFC) or [GitHub](https://github.com/vercel/next-learn).

# Getting Started

## [Creating a new project](https://nextjs.org/learn/dashboard-app/getting-started#creating-a-new-project)

To create a Next.js app, open your terminal, [`cd`](https://developer.mozilla.org/en-US/docs/Learn/Tools\_and\_testing/Understanding\_client-side\_tools/Command\_line#basic\_built-in\_terminal\_commands) into the folder you'd like to keep your project, and run the following command:

Terminal

```

npx create-next-app@latest nextjs-dashboard --use-npm --example "https://github.com/vercel/next-learn/tree/main/dashboard/starter-example"

```

This command uses [`create-next-app`](https://nextjs.org/docs/app/api-reference/create-next-app), a Command Line Interface (CLI) tool that sets up a Next.js application for you. In the command above, you're also using the `--example` flag with the [starter example](https://github.com/vercel/next-learn/tree/main/dashboard/starter-example) for this course.

## [Exploring the project](https://nextjs.org/learn/dashboard-app/getting-started#exploring-the-project)

Unlike tutorials that have you write code from scratch, much of the code for this course is already written for you. This better reflects real-world development, where you'll likely be working with existing codebases.

Our goal is to help you focus on learning the main features of Next.js, without having to write \_all\_ the application code.

After installation, open the project in your code editor and navigate to `nextjs-dashboard`.

Terminal

```

cd nextjs-dashboard

```

Let's spend some time exploring the project.

### [Folder structure](https://nextjs.org/learn/dashboard-app/getting-started#folder-structure)

You'll notice that the project has the following folder structure:

![Folder structure of the dashboard project, showing the main folders and files: app, public, and config files.](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Flearn-folder-structure.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

- \*\*`/app`\*\*: Contains all the routes, components, and logic for your application, this is where you'll be mostly working from.

- \*\*`/app/lib`\*\*: Contains functions used in your application, such as reusable utility functions and data fetching functions.

- \*\*`/app/ui`\*\*: Contains all the UI components for your application, such as cards, tables, and forms. To save time, we've pre-styled these components for you.

- \*\*`/public`\*\*: Contains all the static assets for your application, such as images.

- \*\*`/scripts`\*\*: Contains a seeding script that you'll use to populate your database in a later chapter.

- \*\*Config Files\*\*: You'll also notice config files such as `next.config.js` at the root of your application. Most of these files are created and pre-configured when you start a new project using `create-next-app`. You will not need to modify them in this course.

Feel free to explore these folders, and don't worry if you don't understand everything the code is doing yet.

### [Placeholder data](https://nextjs.org/learn/dashboard-app/getting-started#placeholder-data)

When you're building user interfaces, it helps to have some placeholder data. If a database or API is not yet available, you can:

- Use placeholder data in JSON format or as JavaScript objects.

- Use a 3rd party service like [mockAPI](https://mockapi.io/).

For this project, we've provided some placeholder data in `app/lib/placeholder-data.js`. Each JavaScript object in the file represents a table in your database. For example, for the invoices table:

/app/lib/placeholder-data.js

```

const invoices = [ { customer\_id: customers[0].id, amount: 15795, status: 'pending', date: '2022-12-06', }, { customer\_id: customers[1].id, amount: 20348, status: 'pending', date: '2022-11-14', }, // ...];

```

In the chapter on [setting up your database](https://nextjs.org/learn/dashboard-app/setting-up-your-database), you'll use this data to \_seed\_ your database (populate it with some initial data).

### [TypeScript](https://nextjs.org/learn/dashboard-app/getting-started#typescript)

You may also notice most files have a `.ts` or `.tsx` suffix. This is because the project is written in TypeScript. We wanted to create a course that reflects the modern web landscape.

It's okay if you don't know TypeScript - we'll provide the TypeScript code snippets when required.

For now, take a look at the `/app/lib/definitions.ts` file. Here, we manually define the types that will be returned from the database. For example, the invoices table has the following types:

/app/lib/definitions.ts

```

export type Invoice = { id: string; customer\_id: string; amount: number; date: string; // In TypeScript, this is called a string union type. // It means that the "status" property can only be one of the two strings: 'pending' or 'paid'. status: 'pending' | 'paid';};

```

By using TypeScript, you can ensure you don't accidentally pass the wrong data format to your components or database, like passing a `string` instead of a `number` to invoice `amount`.

> \*\*If you're a TypeScript developer:\*\*

>

> - We're manually declaring the data types, but for better type-safety, we recommend [Prisma](https://www.prisma.io/), which automatically generates types based on your database schema.

> - Next.js detects if your project uses TypeScript and automatically installs the necessary packages and configuration. Next.js also comes with a [TypeScript plugin](https://nextjs.org/docs/app/building-your-application/configuring/typescript#typescript-plugin) for your code editor, to help with auto-completion and type-safety.

## [Running the development server](https://nextjs.org/learn/dashboard-app/getting-started#running-the-development-server)

Run `npm i` to install the project's packages.

Terminal

```

npm i

```

Followed by `npm run dev` to start the development server.

Terminal

```

npm run dev

```

`npm run dev` starts your Next.js development server on port `3000`. Let's check to see if it's working. Open [http://localhost:3000](http://localhost:3000/) on your browser. Your home page should look like this:

![Unstyled page with the title 'Acme', a description, and login link.](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Facme-unstyled.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

# CSS Styling

Currently, your home page doesn't have any styles. Let's look at the different ways you can style your Next.js application.

In this chapter...

Here are the topics we’ll cover

How to add a global CSS file to your application.

Two different ways of styling: Tailwind and CSS modules.

How to conditionally add class names with the `clsx` utility package.

## [Global styles](https://nextjs.org/learn/dashboard-app/css-styling#global-styles)

If you look inside the `/app/ui` folder, you'll see a file called `global.css`. You can use this file to add CSS rules to \*\*all\*\* the routes in your application - such as CSS reset rules, site-wide styles for HTML elements like links, and more.

You can import `global.css` in any component in your application, but it's usually good practice to add it to your top-level component. In Next.js, this is the [root layout](https://nextjs.org/docs/app/building-your-application/routing/pages-and-layouts#root-layout-required) (more on this later).

Add global styles to your application by navigating to `/app/layout.tsx` and importing the `global.css` file:

/app/layout.tsx

```

import '@/app/ui/global.css'; export default function RootLayout({ children,}: { children: React.ReactNode;}) { return ( <html lang="en"> <body>{children}</body> </html> );}

```

With the development server still running, save your changes and preview them in the browser. Your home page should now look like this:

![Styled page with the logo 'Acme', a description, and login link.](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fhome-page-with-tailwind.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

But wait a second, you didn't add any CSS rules, where did the styles come from?

If you take a look inside `global.css`, you'll notice some `@tailwind` directives:

/app/ui/global.css

```

@tailwind base;@tailwind components;@tailwind utilities;

```

## [Tailwind](https://nextjs.org/learn/dashboard-app/css-styling#tailwind)

[Tailwind](https://tailwindcss.com/) is a CSS framework that speeds up the development process by allowing you to quickly write [utility classes](https://tailwindcss.com/docs/utility-first) directly in your TSX markup.

In Tailwind, you style elements by adding class names. For example, adding the class `"text-blue-500"` will turn the `<h1>` text blue:

```

<h1 className="text-blue-500">I'm blue!</h1>

```

Although the CSS styles are shared globally, each class is singularly applied to each element. This means if you add or delete an element, you don't have to worry about maintaining separate stylesheets, style collisions, or the size of your CSS bundle growing as your application scales.

When you use `create-next-app` to start a new project, Next.js will ask if you want to use Tailwind. If you select `yes`, Next.js will automatically install the necessary packages and configure Tailwind in your application.

If you look at `/app/page.tsx`, you'll see that we're using Tailwind classes in the example.

/app/page.tsx

```

import AcmeLogo from '@/app/ui/acme-logo';import { ArrowRightIcon } from '@heroicons/react/24/outline';import Link from 'next/link'; export default function Page() { return ( // These are Tailwind classes: <main className="flex min-h-screen flex-col p-6"> <div className="flex h-20 shrink-0 items-end rounded-lg bg-blue-500 p-4 md:h-52"> // ... )}

```

Don't worry if this is your first time using Tailwind. To save time, we've already styled all the components you'll be using.

Let's play with Tailwind! Copy the code below and paste it above the `<p>` element in `/app/page.tsx`:

/app/page.tsx

```

<div className="h-0 w-0 border-b-[30px] border-l-[20px] border-r-[20px] border-b-black border-l-transparent border-r-transparent"/>

```

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

What shape do you see when using the code snippet above?

If you prefer writing traditional CSS rules or keeping your styles separate from your JSX - CSS Modules are a great alternative.

## [CSS Modules](https://nextjs.org/learn/dashboard-app/css-styling#css-modules)

[CSS Modules](https://nextjs.org/docs/basic-features/built-in-css-support) allow you to scope CSS to a component by automatically creating unique class names, so you don't have to worry about style collisions as well.

We'll continue using Tailwind in this course, but let's take a moment to see how you can achieve the same results from the quiz above using CSS modules.

Inside `/app/ui`, create a new file called `home.module.css` and add the following CSS rules:

/app/ui/home.module.css

```

.shape { height: 0; width: 0; border-bottom: 30px solid black; border-left: 20px solid transparent; border-right: 20px solid transparent;}

```

Then, inside your `/app/page.tsx` file import the styles and replace the tailwind class names from the `<div>` you've added with `styles.shape`:

/app/page.tsx

```

import styles from '@/app/ui/home.module.css'; //...<div className="flex flex-col justify-center gap-6 rounded-lg bg-gray-50 px-6 py-10 md:w-2/5 md:px-20"> <div className={styles.shape}></div>;// ...

```

Save your changes and preview them in the browser. You should see the same shape as before.

Tailwind and CSS modules are the two most common ways of styling Next.js applications. Whether you use one or the other is a matter of preference - you can even use both in the same application!

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

What is one benefit of using CSS modules?

Increase the global scope of CSS classes, making them easier to manage across different files.

Provide a way to make CSS classes locally scoped to components by default, reducing the risk of styling conflicts.

Automatically compress and minify CSS files for faster page loading.

Check Answer

## [Using the `clsx` library to toggle class names](https://nextjs.org/learn/dashboard-app/css-styling#using-the-clsx-library-to-toggle-class-names)

There may be cases where you may need to conditionally style an element based on state or some other condition.

[`clsx`](https://www.npmjs.com/package/clsx) is a library that lets you toggle class names easily. We recommend taking a look at [documentation](https://github.com/lukeed/clsx) for more details, but here's the basic usage:

- Suppose that you want to create an `InvoiceStatus` component which accepts `status`. The status can be `'pending'` or `'paid'`.

- If it's `'paid'`, you want the color to be green. If it's `'pending'`, you want the color to be gray.

You can use `clsx` to conditionally apply the classes, like this:

/app/ui/invoices/status.tsx

```

import clsx from 'clsx'; export default function InvoiceStatus({ status }: { status: string }) { return ( <span className={clsx( 'inline-flex items-center rounded-full px-2 py-1 text-sm', { 'bg-gray-100 text-gray-500': status === 'pending', 'bg-green-500 text-white': status === 'paid', }, )} > // ...)}

```

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

Search for "clsx" in your code editor, what components use it to conditionally apply class names?

## [Other styling solutions](https://nextjs.org/learn/dashboard-app/css-styling#other-styling-solutions)

In addition to the approaches we've discussed, you can also style your Next.js application with:

- Sass which allows you to import `.css` and `.scss` files.

- CSS-in-JS libraries such as [styled-jsx](https://github.com/vercel/styled-jsx), [styled-components](https://github.com/vercel/next.js/tree/canary/examples/with-styled-components), and [emotion](https://github.com/vercel/next.js/tree/canary/examples/with-emotion).

Take a look at the [CSS documentation](https://nextjs.org/docs/app/building-your-application/styling) for more information.

# Optimizing Fonts and Images

In the previous chapter, you learned how to style your Next.js application. Let's continue working on your home page by adding a custom font and a hero image.

In this chapter...

Here are the topics we’ll cover

How to add custom fonts with `next/font`.

How to add images with `next/image`.

How fonts and images are optimized in Next.js.

## [Why optimize fonts?](https://nextjs.org/learn/dashboard-app/optimizing-fonts-images#why-optimize-fonts)

Fonts play a significant role in the design of a website, but using custom fonts in your project can affect performance if the font files need to be fetched and loaded.

[Cumulative Layout Shift](https://web.dev/cls/) is a metric used by Google to evaluate the performance and user experience of a website. With fonts, layout shift happens when the browser initially renders text in a fallback or system font and then swaps it out for a custom font once it has loaded. This swap can cause the text size, spacing, or layout to change, shifting elements around it.

![Mock UI showing initial load of a page, followed by a layout shift as the custom font loads.](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Ffont-layout-shift.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

Next.js automatically optimizes fonts in the application when you use the `next/font` module. It downloads font files at build time and hosts them with your other static assets. This means when a user visits your application, there are no additional network requests for fonts which would impact performance.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

How does Next.js optimize fonts?

It hosts font files with other static assets so that there are no additional network requests.

## [Adding a primary font](https://nextjs.org/learn/dashboard-app/optimizing-fonts-images#adding-a-primary-font)

Let's add a custom Google font to your application to see how this works!

In your `/app/ui` folder, create a new file called `fonts.ts`. You'll use this file to keep the fonts that will be used throughout your application.

Import the `Inter` font from the `next/font/google` module - this will be your primary font. Then, specify what [subset](https://fonts.google.com/knowledge/glossary/subsetting) you'd like to load. In this case, `'latin'`:

/app/ui/fonts.ts

```

import { Inter } from 'next/font/google'; export const inter = Inter({ subsets: ['latin'] });

```

Finally, add the font to the `<body>` element in `/app/layout.tsx`:

/app/layout.tsx

```

import '@/app/ui/global.css';import { inter } from '@/app/ui/fonts'; export default function RootLayout({ children,}: { children: React.ReactNode;}) { return ( <html lang="en"> <body className={`${inter.className} antialiased`}>{children}</body> </html> );}

```

By adding `Inter` to the `<body>` element, the font will be applied throughout your application. Here, you're also adding the Tailwind [`antialiased`](https://tailwindcss.com/docs/font-smoothing) class which smooths out the font. It's not necessary to use this class, but it adds a nice touch.

Navigate to your browser, open dev tools and select the `body` element. You should see `Inter` and `Inter\_Fallback` are now applied under styles.

## [Practice: Adding a secondary font](https://nextjs.org/learn/dashboard-app/optimizing-fonts-images#practice-adding-a-secondary-font)

You can also add fonts to specific elements of your application.

Now it's your turn! In your `fonts.ts` file, import a secondary font called `Lusitana` and pass it to the `<p>` element in your `/app/page.tsx` file. In addition to specifying a subset like you did before, you'll also need to specify the font \*\*weight\*\*.

Once you're ready, expand the code snippet below to see the solution.

> \*\*Hints:\*\*

>

> - If you're unsure what weight options to pass to a font, check the TypeScript errors in your code editor.

> - Visit the [Google Fonts](https://fonts.google.com/) website and search for `Lusitana` to see what options are available.

> - See the documentation for [adding multiple fonts](https://nextjs.org/docs/app/building-your-application/optimizing/fonts#using-multiple-fonts) and the [full list of options](https://nextjs.org/docs/app/api-reference/components/font#font-function-arguments).

Reveal the solution

Finally, the `<AcmeLogo />` component also uses Lusitana. It was commented out to prevent errors, you can now uncomment it:

/app/page.tsx

```

// ... export default function Page() { return ( <main className="flex min-h-screen flex-col p-6"> <div className="flex h-20 shrink-0 items-end rounded-lg bg-blue-500 p-4 md:h-52"> <AcmeLogo /> {/\* ... \*/} </div> </main> );}

```

Great, you've added two custom fonts to your application! Next, let's add a hero image to the home page.

## [Why optimize images?](https://nextjs.org/learn/dashboard-app/optimizing-fonts-images#why-optimize-images)

Next.js can serve \*\*static assets\*\*, like images, under the top-level [`/public`](https://nextjs.org/docs/app/building-your-application/optimizing/static-assets) folder. Files inside `/public` can be referenced in your application.

With regular HTML, you would add an image as follows:

```

<img src="/hero.png" alt="Screenshots of the dashboard project showing desktop version"/>

```

However, this means you have to manually:

- Ensure your image is responsive on different screen sizes.

- Specify image sizes for different devices.

- Prevent layout shift as the images load.

- Lazy load images that are outside the user's viewport.

Image Optimization is a large topic in web development that could be considered a specialization in itself. Instead of manually implementing these optimizations, you can use the `next/image` component to automatically optimize your images.

## [The `<Image>` component](https://nextjs.org/learn/dashboard-app/optimizing-fonts-images#the-image-component)

The `<Image>` Component is an extension of the HTML `<img>` tag, and comes with automatic image optimization, such as:

- Preventing layout shift automatically when images are loading.

- Resizing images to avoid shipping large images to devices with a smaller viewport.

- Lazy loading images by default (images load as they enter the viewport).

- Serving images in modern formats, like [WebP](https://developer.mozilla.org/en-US/docs/Web/Media/Formats/Image\_types#webp) and [AVIF](https://developer.mozilla.org/en-US/docs/Web/Media/Formats/Image\_types#avif\_image), when the browser supports it.

## [Adding the desktop hero image](https://nextjs.org/learn/dashboard-app/optimizing-fonts-images#adding-the-desktop-hero-image)

Let's use the `<Image>` component. If you look inside the `/public` folder, you'll see there are two images: `hero-desktop.png` and `hero-mobile.png`. These two images are completely different, and they'll be shown depending if the user's device is a desktop or mobile.

In your `/app/page.tsx` file, import the component from [`next/image`](https://nextjs.org/docs/api-reference/next/image). Then, add the image under the comment:

/app/page.tsx

```

import AcmeLogo from '@/app/ui/acme-logo';import { ArrowRightIcon } from '@heroicons/react/24/outline';import Link from 'next/link';import { lusitana } from '@/app/ui/fonts';import Image from 'next/image'; export default function Page() { return ( // ... <div className="flex items-center justify-center p-6 md:w-3/5 md:px-28 md:py-12"> {/\* Add Hero Images Here \*/} <Image src="/hero-desktop.png" width={1000} height={760} className="hidden md:block" alt="Screenshots of the dashboard project showing desktop version" /> </div> //... );}

```

Here, you're setting the `width` to `1000` and `height` to `760` pixels. It's good practice to set the `width` and `height` of your images to avoid layout shift, these should be an aspect ratio \*\*identical\*\* to the source image.

You'll also notice the class `hidden` to remove the image from the DOM on mobile screens, and `md:block` to show the image on desktop screens.

This is what your home page should look like now:

![Styled home page with a custom font and hero image](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fhome-page-with-hero.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

## [Practice: Adding the mobile hero image](https://nextjs.org/learn/dashboard-app/optimizing-fonts-images#practice-adding-the-mobile-hero-image)

Now it's your turn! Under the image you've just added, add another `<Image>` component for `hero-mobile.png`.

- The image should have a `width` of `560` and `height` of `620` pixels.

- It should be shown on mobile screens, and hidden on desktop - you can use dev tools to check if the desktop and mobile images are swapped correctly.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

True or False: Images without dimensions and web fonts are common causes of layout shift.

## [Recommended reading](https://nextjs.org/learn/dashboard-app/optimizing-fonts-images#recommended-reading)

There's a lot more to learn about these topics, including optimizing remote images and using local font files. If you'd like to dive deeper into fonts and images, see:

- [Image Optimization Docs](https://nextjs.org/docs/app/building-your-application/optimizing/images)

- [Font Optimization Docs](https://nextjs.org/docs/app/building-your-application/optimizing/fonts)

- [Improving Web Performance with Images (MDN)](https://developer.mozilla.org/en-US/docs/Learn/Performance/Multimedia)

- [Web Fonts (MDN)](https://developer.mozilla.org/en-US/docs/Learn/CSS/Styling\_text/Web\_fonts)

# Creating Layouts and Pages

So far, your application only has a home page. Let's learn how you can create more routes with \*\*layouts\*\* and \*\*pages\*\*.

In this chapter...

Here are the topics we’ll cover

Create the `dashboard` routes using file-system routing.

Understand the role of folders and files when creating new route segments.

Create a nested layout that can be shared between multiple dashboard pages.

Understand what colocation, partial rendering, and the root layout are.

## [Nested routing](https://nextjs.org/learn/dashboard-app/creating-layouts-and-pages#nested-routing)

Next.js uses file-system routing where \*\*folders\*\* are used to create nested routes. Each folder represents a \*\*route segment\*\* that maps to a \*\*URL segment\*\*.

![Diagram showing how folders map to URL segments](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Ffolders-to-url-segments.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

You can create separate UIs for each route using `layout.tsx` and `page.tsx` files.

`page.tsx` is a special Next.js file that exports a React component, and it's required for the route to be accessible. In your application, you already have a page file: `/app/page.tsx` - this is the home page associated with the route `/`.

To create a nested route, you can nest folders inside each other and add `page.tsx` files inside them. For example:

![Diagram showing how adding a folder called dashboard creates a new route '/dashboard'](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fdashboard-route.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

`/app/dashboard/page.tsx` is associated with the `/dashboard` path. Let's create the page to see how it works!

## [Creating the dashboard page](https://nextjs.org/learn/dashboard-app/creating-layouts-and-pages#creating-the-dashboard-page)

Create a new folder called `dashboard` inside `/app`. Then, create a new `page.tsx` file inside the `dashboard` folder with the following content:

/app/dashboard/page.tsx

```

export default function Page() { return <p>Dashboard Page</p>;}

```

Now, make sure that the development server is running and visit [http://localhost:3000/dashboard](http://localhost:3000/dashboard). You should see the "Dashboard Page" text.

This is how you can create different pages in Next.js: create a new route segment using a folder, and add a `page` file inside it.

By having a special name for `page` files, Next.js allows you to [colocate](https://nextjs.org/docs/app/building-your-application/routing#colocation) UI components, test files, and other related code with your routes. Only the content inside the `page` file will be publicly accessible. For example, the `/ui` and `/lib` folders are \_colocated\_ inside the `/app` folder along with your routes.

## [Practice: Creating the dashboard pages](https://nextjs.org/learn/dashboard-app/creating-layouts-and-pages#practice-creating-the-dashboard-pages)

Let's practice creating more routes. In your dashboard, create two more pages:

1. \*\*Customers Page\*\*: The page should be accessible on [http://localhost:3000/dashboard/customers](http://localhost:3000/dashboard/customers). For now, it should return a `<p>Customers Page</p>` element.

2. \*\*Invoices Page\*\*: The invoices page should be accessible on [http://localhost:3000/dashboard/invoices](http://localhost:3000/dashboard/invoices). For now, also return a `<p>Invoices Page</p>` element.

Spend some time tackling this exercise, and when you're ready, expand the toggle below for the solution:

Reveal the solution

## [Creating the dashboard layout](https://nextjs.org/learn/dashboard-app/creating-layouts-and-pages#creating-the-dashboard-layout)

Dashboards have some sort of navigation that is shared across multiple pages. In Next.js, you can use a special `layout.tsx` file to create UI that is shared between multiple pages. Let's create a layout for the dashboard pages!

Inside the `/dashboard` folder, add a new file called `layout.tsx` and paste the following code:

/app/dashboard/layout.tsx

```

import SideNav from '@/app/ui/dashboard/sidenav'; export default function Layout({ children }: { children: React.ReactNode }) { return ( <div className="flex h-screen flex-col md:flex-row md:overflow-hidden"> <div className="w-full flex-none md:w-64"> <SideNav /> </div> <div className="flex-grow p-6 md:overflow-y-auto md:p-12">{children}</div> </div> );}

```

A few things are going on in this code, so let's break it down:

First, you're importing the `<SideNav />` component into your layout. Any components you import into this file will be part of the layout.

The `<Layout />` component receives a `children` prop. This child can either be a page or another layout. In your case, the pages inside `/dashboard` will automatically be nested inside a `<Layout />` like so:

![Folder structure with dashboard layout nesting the dashboard pages as children](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fshared-layout.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

Check that everything is working correctly by saving your changes and checking your localhost. You should see the following:

![Dashboard page with a sidenav and a main content area](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fshared-layout-page.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

One benefit of using layouts in Next.js is that on navigation, only the page components update while the layout won't re-render. This is called [partial rendering](https://nextjs.org/docs/app/building-your-application/routing/linking-and-navigating#3-partial-rendering):

![Folder structure showing the dashboard layout nesting the dashboard pages, but only the pages UI swap on navigation](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fpartial-rendering-dashboard.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

## [Root layout](https://nextjs.org/learn/dashboard-app/creating-layouts-and-pages#root-layout)

In Chapter 3, you imported the `Inter` font into another layout: `/app/layout.tsx`. As a reminder:

/app/layout.tsx

```

import '@/app/ui/global.css';import { inter } from '@/app/ui/fonts'; export default function RootLayout({ children,}: { children: React.ReactNode;}) { return ( <html lang="en"> <body className={`${inter.className} antialiased`}>{children}</body> </html> );}

```

This is called a [root layout](https://nextjs.org/docs/app/building-your-application/routing/pages-and-layouts#root-layout-required) and is required. Any UI you add to the root layout will be shared across \*\*all\*\* pages in your application. You can use the root layout to modify your `<html>` and `<body>` tags, and add metadata (you'll learn more about metadata in [a later chapter](https://nextjs.org/learn/dashboard-app/adding-metadata)).

Since the new layout you've just created (`/app/dashboard/layout.tsx`) is unique to the dashboard pages, you don't need to add any UI to the root layout above.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

What is the purpose of the layout file in Next.js?

# Navigating Between Pages

In the previous chapter, you created the dashboard layout and pages. Now, let's add some links to allow users to navigate between the dashboard routes.

In this chapter...

Here are the topics we’ll cover

How to use the `next/link` component.

How to show an active link with the `usePathname()` hook.

How navigation works in Next.js.

## [Why optimize navigation?](https://nextjs.org/learn/dashboard-app/navigating-between-pages#why-optimize-navigation)

To link between pages, you'd traditionally use the `<a>` HTML element. At the moment, the sidebar links use `<a>` elements, but notice what happens when you navigate between the home, invoices, and customers pages on your browser.

Did you see it?

There's a full page refresh on each page navigation!

## [The `<Link>` component](https://nextjs.org/learn/dashboard-app/navigating-between-pages#the-link-component)

In Next.js, you can use the `<Link />` Component to link between pages in your application. `<Link>` allows you to do [client-side navigation](https://nextjs.org/docs/app/building-your-application/routing/linking-and-navigating#how-routing-and-navigation-works) with JavaScript.

To use the `<Link />` component, open `/app/ui/dashboard/nav-links.tsx`, and import the `Link` component from [`next/link`](https://nextjs.org/docs/app/api-reference/components/link). Then, replace the `<a>` tag with `<Link>`:

/app/ui/dashboard/nav-links.tsx

```

import { UserGroupIcon, HomeIcon, DocumentDuplicateIcon,} from '@heroicons/react/24/outline';import Link from 'next/link'; // ... export default function NavLinks() { return ( <> {links.map((link) => { const LinkIcon = link.icon; return ( <Link key={link.name} href={link.href} className="flex h-[48px] grow items-center justify-center gap-2 rounded-md bg-gray-50 p-3 text-sm font-medium hover:bg-sky-100 hover:text-blue-600 md:flex-none md:justify-start md:p-2 md:px-3" > <LinkIcon className="w-6" /> <p className="hidden md:block">{link.name}</p> </Link> ); })} </> );}

```

As you can see, the `Link` component is similar to using `<a>` tags, but instead of `<a href="…">`, you use `<Link href="…">`.

Save your changes and check to see if it works in your localhost. You should now be able to navigate between the pages without seeing a full refresh. Although parts of your application are rendered on the server, there's no full page refresh, making it feel like a web app. Why is that?

### [Automatic code-splitting and prefetching](https://nextjs.org/learn/dashboard-app/navigating-between-pages#automatic-code-splitting-and-prefetching)

To improve the navigation experience, Next.js automatically code splits your application by route segments. This is different from a traditional React [SPA](https://developer.mozilla.org/en-US/docs/Glossary/SPA), where the browser loads all your application code on initial load.

Splitting code by routes means that pages become isolated. If a certain page throws an error, the rest of the application will still work.

Futhermore, in production, whenever [`<Link>`](https://nextjs.org/docs/api-reference/next/link) components appear in the browser's viewport, Next.js automatically \*\*prefetches\*\* the code for the linked route in the background. By the time the user clicks the link, the code for the destination page will already be loaded in the background, and this is what makes the page transition near-instant!

Learn more about [how navigation works](https://nextjs.org/docs/app/building-your-application/routing/linking-and-navigating#how-routing-and-navigation-works).

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

What does Next.js do when a <Link> component appears in the browser’s viewport in a production environment?

## [Pattern: Showing active links](https://nextjs.org/learn/dashboard-app/navigating-between-pages#pattern-showing-active-links)

A common UI pattern is to show an active link to indicate to the user what page they are currently on. To do this, you need to get the user's current path from the URL. Next.js provides a hook called [`usePathname()`](https://nextjs.org/docs/app/api-reference/functions/use-pathname) that you can use to check the path and implement this pattern.

Since [`usePathname()`](https://nextjs.org/docs/app/api-reference/functions/use-pathname) is a hook, you'll need to turn `nav-links.tsx` into a Client Component. Add React's `"use client"` directive to the top of the file, then import `usePathname()` from `next/navigation`:

/app/ui/dashboard/nav-links.tsx

```

'use client'; import { UserGroupIcon, HomeIcon, InboxIcon,} from '@heroicons/react/24/outline';import Link from 'next/link';import { usePathname } from 'next/navigation'; // ...

```

Next, assign the path to a variable called `pathname` inside your `<NavLinks />` component:

/app/ui/dashboard/nav-links.tsx

```

export default function NavLinks() { const pathname = usePathname(); // ...}

```

You can use the `clsx` library introduced in the chapter on [CSS styling](https://nextjs.org/learn/dashboard-app/css-styling) to conditionally apply class names when the link is active. When `link.href` matches the `pathname`, the link should displayed with blue text and a light blue background.

Here's the final code for `nav-links.tsx`:

/app/ui/dashboard/nav-links.tsx

```

'use client'; import { UserGroupIcon, HomeIcon, DocumentDuplicateIcon,} from '@heroicons/react/24/outline';import Link from 'next/link';import { usePathname } from 'next/navigation';import clsx from 'clsx'; // ... export default function NavLinks() { const pathname = usePathname(); return ( <> {links.map((link) => { const LinkIcon = link.icon; return ( <Link key={link.name} href={link.href} className={clsx( 'flex h-[48px] grow items-center justify-center gap-2 rounded-md bg-gray-50 p-3 text-sm font-medium hover:bg-sky-100 hover:text-blue-600 md:flex-none md:justify-start md:p-2 md:px-3', { 'bg-sky-100 text-blue-600': pathname === link.href, }, )} > <LinkIcon className="w-6" /> <p className="hidden md:block">{link.name}</p> </Link> ); })} </> );}

```

Save and check your localhost. You should now see the active link highlighted in blue.

5

## You've Completed Chapter 5

You've learned how to link between pages and leverage client-side navigation in Next.js.

# Setting Up Your Database

Before you can continue working on your dashboard, you'll need some data. In this chapter, you'll be setting up a PostgreSQL database using `@vercel/postgres`. If you're already familiar with PostgreSQL and would prefer to use your own provider, you can skip this chapter and set it up on your own. Otherwise, let's continue!

In this chapter...

Here are the topics we’ll cover

Push your project to GitHub.

Set up a Vercel account and link your GitHub repo for instant previews and deployments.

Create and link your project to a Postgres database.

Seed the database with initial data.

## [Create a GitHub repository](https://nextjs.org/learn/dashboard-app/setting-up-your-database#create-a-github-repository)

To start, let's push your repository to Github if you haven't done so already. This will make it easier to set up your database and deploy.

If you need help setting up your repository, take a look at [this guide on GitHub](https://help.github.com/en/github/getting-started-with-github/create-a-repo).

> \*\*Good to know:\*\*

>

> - You can also use other Git provider like GitLab or Bitbucket.

> - If you're new to GitHub, we recommend the [GitHub Desktop App](https://desktop.github.com/) for a simplified development workflow.

## [Create a Vercel account](https://nextjs.org/learn/dashboard-app/setting-up-your-database#create-a-vercel-account)

Visit [vercel.com/signup](https://vercel.com/signup) to create an account. Choose the free "hobby" plan. Select \*\*Continue with GitHub\*\* to connect your GitHub and Vercel accounts.

## [Connect and deploy your project](https://nextjs.org/learn/dashboard-app/setting-up-your-database#connect-and-deploy-your-project)

Next, you'll be taken to this screen where you can select and \*\*import\*\* the GitHub repository you've just created:

![Screenshot of Vercel Dashboard, showing the import project screen with a list of the user's GitHub Repositories](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fimport-git-repo.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

Name your project and click \*\*Deploy\*\*.

![Deployment screen showing the project name field and a deploy button](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fconfigure-project.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

Hooray! 🎉 Your project is now deployed.

![Project overview screen showing the project name, domain, and deployment status](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fdeployed-project.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

By connecting your GitHub repository, whenever you push changes to your \*\*main\*\* branch, Vercel will automatically redeploy your application with no configuration needed. When opening pull requests, you'll also have [instant previews](https://vercel.com/docs/deployments/preview-deployments#preview-urls) which allow you to catch deployment errors early and share a preview of your project with team members for feedback.

## [Create a Postgres database](https://nextjs.org/learn/dashboard-app/setting-up-your-database#create-a-postgres-database)

Next, to set up a database, click \*\*Continue to Dashboard\*\* and select the \*\*Storage\*\* tab from your project dashboard. Select \*\*Connect Store\*\* → \*\*Create New\*\* → \*\*Postgres\*\* → \*\*Continue\*\*.

![Connect Store screen showing the Postgres option along with KV, Blob and Edge Config](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fcreate-database.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

Accept the terms, assign a name to your database, and ensure your database region is set to \*\*Washington D.C (iad1)\*\* - this is also the [default region](https://vercel.com/docs/functions/serverless-functions/regions#select-a-default-serverless-region) for all new Vercel projects. By placing your database in the same region or close to your application code, you can reduce [latency](https://developer.mozilla.org/en-US/docs/Web/Performance/Understanding\_latency) for data requests.

![Database creation modal showing the database name and region](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fdatabase-region.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

> \*\*Good to know\*\*: You cannot change the database region once it has been initalized. If you wish to use a different [region](https://vercel.com/docs/storage/vercel-postgres/limits#supported-regions), you should set it before creating a database.

Once connected, navigate to the `.env.local` tab, click \*\*Show secret\*\* and \*\*Copy Snippet\*\*. Make sure you reveal the secrets before copying them.

![The .env.local tab showing the hidden database secrets](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fdatabase-dashboard.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

Navigate to your code editor and rename the `.env.example` file to \*\*`.env`\*\*. Paste in the copied contents from Vercel.

\*\*Important:\*\* Go to your `.gitignore` file and make sure `.env` is in the ignored files to prevent your database secrets from being exposed when you push to GitHub.

Finally, run `npm i @vercel/postgres` in your terminal to install the [Vercel Postgres SDK](https://vercel.com/docs/storage/vercel-postgres/sdk).

## [Seed your database](https://nextjs.org/learn/dashboard-app/setting-up-your-database#seed-your-database)

Now that your database has been created, let's seed it with some initial data. This will allow you to have some data to work with as you build the dashboard.

In the `/scripts` folder of your project, there's a file called `seed.js`. This script contains the instructions for creating and seeding the `invoices`, `customers`, `user`, `revenue` tables.

Don't worry if you don't understand everything the code is doing, but to give you an overview, the script uses \*\*SQL\*\* to create the tables, and the data from `placeholder-data.js` file to populate them after they've been created.

Next, in your `package.json` file, add the following line to your scripts:

/package.json

```

"scripts": { "build": "next build", "dev": "next dev", "start": "next start", "seed": "node -r dotenv/config ./scripts/seed.js"},

```

This is the command that will execute `seed.js`.

Now, run `npm run seed`. You should see some `console.log` messages in your terminal to let you know the script is running.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

What is 'seeding' in the context of databases?

> \*\*Troubleshooting\*\*:

>

> - Make sure to reveal your database secrets before copying it into your `.env` file.

> - The script uses `bcrypt` to hash the user's password, if `bcrypt` isn't compatible with your environment, you can update the script to use [`bcryptjs`](https://www.npmjs.com/package/bcryptjs) instead.

> - If you run into any issues while seeding your database and want to run the script again, you can drop any existing tables by running `DROP TABLE tablename` in your database query interface. See the [executing queries section](https://nextjs.org/learn/dashboard-app/setting-up-your-database#executing-queries) below for more details. But be careful, this command will delete the tables and all their data. It's ok to do this with your example app since you're working with placeholder data, but you shouldn't run this command in a production app.

> - If you continue to experience issues while seeding your Verce Postgres database, please open a [discussion on GitHub](https://github.com/vercel/next-learn/issues).

## [Exploring your database](https://nextjs.org/learn/dashboard-app/setting-up-your-database#exploring-your-database)

Let's see what your database looks like. Go back to Vercel, and click \*\*Data\*\* on the sidenav.

In this section, you'll find the four new tables: users, customers, invoices, and revenue.

![Database screen showing dropdown list with four tables: users, customers, invoices, and revenue](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fdatabase-tables.png&w=2048&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

By selecting each table, you can view its records and ensure the entries align with the data from `placeholder-data.js` file.

## [Executing queries](https://nextjs.org/learn/dashboard-app/setting-up-your-database#executing-queries)

You can switch to the "query" tab to interact with your database. This section supports standard SQL commands. For instance, inputting `DROP TABLE customers` will delete "customers" table along with all its data - \*\*\_so be careful\_\*\*!

Let's run your first database query. Paste and run the following SQL code into the Vercel interface:

```

SELECT invoices.amount, customers.nameFROM invoicesJOIN customers ON invoices.customer\_id = customers.idWHERE invoices.amount = 666;

```

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

Which customer does this invoice belong to?

## You've Completed Chapter 6

With your database now set up and integrated, you can continue building your application.

7: Fetching Data

Let's discuss the different ways you can fetch data from your database, including using APIs, SQL, and alternatives.

[Start Chapter 7](https://nextjs.org/learn/dashboard-app/fetching-data)

# Fetching Data

Now that you've created and seeded your database, let's discuss the different ways you can fetch data for your application, and build out your dashboard overview page.

In this chapter...

Here are the topics we’ll cover

Learn about some approaches to fetching data: APIs, ORMs, SQL, etc.

How Server Components can help you access back-end resources more securely.

What network waterfalls are.

How to implement parallel data fetching using a JavaScript Pattern.

## [Choosing how to fetch data](https://nextjs.org/learn/dashboard-app/fetching-data#choosing-how-to-fetch-data)

### [API layer](https://nextjs.org/learn/dashboard-app/fetching-data#api-layer)

APIs are an intermediary layer between your application code and database. There are a few cases where you might use an API:

- If you're using 3rd party services that provide an API.

- If you're fetching data from the client, you want to have an API layer that runs on the server to avoid exposing your database secrets to the client.

In Next.js, you can create API endpoints using [Route Handlers](https://nextjs.org/docs/app/building-your-application/routing/route-handlers).

### [Database queries](https://nextjs.org/learn/dashboard-app/fetching-data#database-queries)

When you're creating a full-stack application, you'll also need to write logic to interact with your database. For [relational databases](https://aws.amazon.com/relational-database/) like Postgres, you can do this with SQL, or an [ORM](https://vercel.com/docs/storage/vercel-postgres/using-an-orm#) like [Prisma](https://www.prisma.io/).

There are a few cases where you have to write database queries:

- When creating your API endpoints, you need to write logic to interact with your database.

- If you are using React Server Components (fetching data on the server), you can skip the API layer, and query your database directly without risking exposing your database secrets to the client.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

In which of these scenarios should you not query your database directly?

### [Using Server Components to fetch data](https://nextjs.org/learn/dashboard-app/fetching-data#using-server-components-to-fetch-data)

By default, Next.js applications use \*\*React Server Components\*\*. Fetching data with Server Components is a relatively new approach and there are a few benefits of using them:

- Server Components support promises, providing a simpler solution for asynchronous tasks like data fetching. You can use `async/await` syntax without reaching out for `useEffect`, `useState` or data fetching libraries.

- Server Components execute on the server, so you can keep expensive data fetches and logic on the server and only send the result to the client.

- As mentioned before, since Server Components execute on the server, you can query the database directly without an additional API layer.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

What's one advantage of using React Server Components to fetch data?

### [Using SQL](https://nextjs.org/learn/dashboard-app/fetching-data#using-sql)

For your dashboard project, you'll write database queries using the [Vercel Postgres SDK](https://vercel.com/docs/storage/vercel-postgres/sdk) and SQL. There are a few reasons why we'll be using SQL:

- SQL is the industry standard for querying relational databases (e.g. ORMs generate SQL under the hood).

- Having a basic understanding of SQL can help you understand the fundamentals of relational databases, allowing you to apply your knowledge to other tools.

- SQL is versatile, allowing you to fetch and manipulate specific data.

- The Vercel Postgres SDK provides protection against [SQL injections](https://vercel.com/docs/storage/vercel-postgres/sdk#preventing-sql-injections).

Don't worry if you haven't used SQL before - we have provided the queries for you.

Go to `/app/lib/data.ts`, here you'll see that we're importing the [`sql`](https://vercel.com/docs/storage/vercel-postgres/sdk#sql) function from `@vercel/postgres`. This function allows you to query your database:

/app/lib/data.ts

```

import { sql } from '@vercel/postgres'; // ...

```

You can call `sql` inside any Server Component. But to allow you to navigate the components more easily, we've kept all the data queries in the `data.ts` file, and you can import them into the components.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

What does SQL allow you to do in terms of fetching data?

> \*\*Note:\*\* If you used your own database provider in Chapter 6, you'll need to update the database queries to work with your provider. You can find the queries in `/app/lib/data.ts`.

## [Fetching data for the dashboard overview page](https://nextjs.org/learn/dashboard-app/fetching-data#fetching-data-for-the-dashboard-overview-page)

Now that you understand different ways of fetching data, let's fetch data for the dashboard overview page. Navigate to `/app/dashboard/page.tsx`, paste the following code, and spend some time exploring it:

/app/dashboard/page.tsx

```

import { Card } from '@/app/ui/dashboard/cards';import RevenueChart from '@/app/ui/dashboard/revenue-chart';import LatestInvoices from '@/app/ui/dashboard/latest-invoices';import { lusitana } from '@/app/ui/fonts'; export default async function Page() { return ( <main> <h1 className={`${lusitana.className} mb-4 text-xl md:text-2xl`}> Dashboard </h1> <div className="grid gap-6 sm:grid-cols-2 lg:grid-cols-4"> {/\* <Card title="Collected" value={totalPaidInvoices} type="collected" /> \*/} {/\* <Card title="Pending" value={totalPendingInvoices} type="pending" /> \*/} {/\* <Card title="Total Invoices" value={numberOfInvoices} type="invoices" /> \*/} {/\* <Card title="Total Customers" value={numberOfCustomers} type="customers" /> \*/} </div> <div className="mt-6 grid grid-cols-1 gap-6 md:grid-cols-4 lg:grid-cols-8"> {/\* <RevenueChart revenue={revenue} /> \*/} {/\* <LatestInvoices latestInvoices={latestInvoices} /> \*/} </div> </main> );}

```

In the code above:

- Page is an \*\*async\*\* component. This allows you to use `await` to fetch data.

- There are also 3 components which receive data: `<Card>`, `<RevenueChart>`, and `<LatestInvoices>`. They are currently commented out to prevent the application from erroring.

## [Fetching data for \*\*`<RevenueChart/>`\*\*](https://nextjs.org/learn/dashboard-app/fetching-data#fetching-data-for-revenuechart)

To fetch data for the `<RevenueChart/>` component, import the `fetchRevenue` function from `data.ts` and call it inside your component:

/app/dashboard/page.tsx

```

import { Card } from '@/app/ui/dashboard/cards';import RevenueChart from '@/app/ui/dashboard/revenue-chart';import LatestInvoices from '@/app/ui/dashboard/latest-invoices';import { lusitana } from '@/app/ui/fonts';import { fetchRevenue } from '@/app/lib/data'; export default async function Page() { const revenue = await fetchRevenue(); // ...}

```

Then, uncomment the `<RevenueChart/>` component, navigate to the component file (`/app/ui/dashboard/revenue-chart.tsx`) and uncomment the code inside it. Check your localhost, you should be able to see a chart that uses `revenue` data.

![Revenue chart showing the total revenue for the last 12 months](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Frecent-revenue.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

Let's continue importing some more data queries!

## [Fetching data for \*\*`<LatestInvoices/>`\*\*](https://nextjs.org/learn/dashboard-app/fetching-data#fetching-data-for-latestinvoices)

For the `<LatestInvoices />` component, we need to get the latest 5 invoices, sorted by date.

You could fetch all the invoices and sort through them using JavaScript. This isn't a problem as our data is small, but as your application grows, it can significantly increase the amount of data transferred on each request and the JavaScript required to sort through it.

Instead of sorting through the latest invoices in-memory, you can use an SQL query to fetch only the last 5 invoices. For example, this is the SQL query from your `data.ts` file:

/app/lib/data.ts

```

// Fetch the last 5 invoices, sorted by dateconst data = await sql<LatestInvoiceRaw>` SELECT invoices.amount, customers.name, customers.image\_url, customers.email FROM invoices JOIN customers ON invoices.customer\_id = customers.id ORDER BY invoices.date DESC LIMIT 5`;

```

In your page, import the `fetchLatestInvoices` function:

/app/dashboard/page.tsx

```

import { Card } from '@/app/ui/dashboard/cards';import RevenueChart from '@/app/ui/dashboard/revenue-chart';import LatestInvoices from '@/app/ui/dashboard/latest-invoices';import { lusitana } from '@/app/ui/fonts';import { fetchRevenue, fetchLatestInvoices } from '@/app/lib/data'; export default async function Page() { const revenue = await fetchRevenue(); const latestInvoices = await fetchLatestInvoices(); // ...}

```

Then, uncomment the `<LatestInvoices />` component. You will also need to uncomment the relevant code in the `<LatestInvoices />` component itself, located at `/app/ui/dashboard/latest-invoices`.

If you visit your localhost, you should see that only the last 5 are returned from the database. Hopefully, you're beginning to see the advantages of querying your database directly!

![Latest invoices component alongside the revenue chart](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Flatest-invoices.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

## [Practice: Fetch data for the `<Card>` components](https://nextjs.org/learn/dashboard-app/fetching-data#practice-fetch-data-for-the-card-components)

Now it's your turn to fetch data for the `<Card>` components. The cards will display the following data:

- Total amount of invoices collected.

- Total amount of invoices pending.

- Total number of invoices.

- Total number of customers.

Again, you might be tempted to fetch all the invoices and customers, and use JavaScript to manipulate the data. For example, you could use `Array.length` to get the total number of invoices and customers:

```

const totalInvoices = allInvoices.length;const totalCustomers = allCustomers.length;

```

But with SQL, you can fetch only the data you need. It's a little longer than using `Array.length`, but it means less data needs to be transferred during the request. This is the SQL alternative:

/app/lib/data.ts

```

const invoiceCountPromise = sql`SELECT COUNT(\*) FROM invoices`;const customerCountPromise = sql`SELECT COUNT(\*) FROM customers`;

```

The function you will need to import is called `fetchCardData`. You will need to destructure the values returned from the function.

> \*\*Hint:\*\*

>

> - Check the card components to see what data they need.

> - Check the `data.ts` file to see what the function returns.

Once you're ready, expand the toggle below for the final code:

Reveal the solution

Great! You've now fetched all the data for the dashboard overview page. Your page should look like this:

![Dashboard page with all the data fetched](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fcomplete-dashboard.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

However... there are two things you need to be aware of:

1. The data requests are unintentionally blocking each other, creating a \*\*request waterfall\*\*.

2. By default, Next.js \*\*prerenders\*\* routes to improve performance, this is called \*\*Static Rendering\*\*. So if your data changes, it won't be reflected in your dashboard.

Let's discuss number 1 in this chapter, then look into detail at number 2 in the next chapter.

## [What are request waterfalls?](https://nextjs.org/learn/dashboard-app/fetching-data#what-are-request-waterfalls)

A "waterfall" refers to a sequence of network requests that depend on the completion of previous requests. In the case of data fetching, each request can only begin once the previous request has returned data.

![Diagram showing time with sequential data fetching and parallel data fetching](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fsequential-parallel-data-fetching.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

For example, we need to wait for `fetchRevenue()` to execute before `fetchLatestInvoices()` can start running, and so on.

/app/dashboard/page.tsx

```

const revenue = await fetchRevenue();const latestInvoices = await fetchLatestInvoices(); // wait for fetchRevenue() to finishconst { numberOfInvoices, numberOfCustomers, totalPaidInvoices, totalPendingInvoices,} = await fetchCardData(); // wait for fetchLatestInvoices() to finish

```

This pattern is not necessarily bad. There may be cases where you want waterfalls because you want a condition to be satisfied before you make the next request. For example, you might want to fetch a user's ID and profile information first. Once you have the ID, you might then proceed to fetch their list of friends. In this case, each request is contingent on the data returned from the previous request.

However, this behavior can also be unintentional and impact performance.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

When might you want to use a waterfall pattern?

## [Parallel data fetching](https://nextjs.org/learn/dashboard-app/fetching-data#parallel-data-fetching)

A common way to avoid waterfalls is to initiate all data requests at the same time - in parallel.

In JavaScript, you can use the [`Promise.all()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Promise/all) or [`Promise.allSettled()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Promise/allSettled) functions to initiate all promises at the same time. For example, in `data.ts`, we're using `Promise.all()` in the `fetchCardData()` function:

/app/lib/data.js

```

export async function fetchCardData() { try { const invoiceCountPromise = sql`SELECT COUNT(\*) FROM invoices`; const customerCountPromise = sql`SELECT COUNT(\*) FROM customers`; const invoiceStatusPromise = sql`SELECT SUM(CASE WHEN status = 'paid' THEN amount ELSE 0 END) AS "paid", SUM(CASE WHEN status = 'pending' THEN amount ELSE 0 END) AS "pending" FROM invoices`; const data = await Promise.all([ invoiceCountPromise, customerCountPromise, invoiceStatusPromise, ]); // ... }}

```

By using this pattern, you can:

- Start executing all data fetches at the same time, which can lead to performance gains.

- Use a native JavaScript pattern that can be applied to any library or framework.

However, there is one \*\*disadvantage\*\* of relying only on this JavaScript pattern: what happens if one data request is slower than all the others?

7

## You've Completed Chapter 7

You've learned about some of the different ways to fetch data in Next.js.

# Static and Dynamic Rendering

In the previous chapter, you fetched data for the Dashboard Overview page. However, we briefly discussed two limitations of the current setup:

1. The data requests are creating an unintentional waterfall.

2. The dashboard is static, so any data updates will not be reflected on your application.

In this chapter...

Here are the topics we’ll cover

What static rendering is and how it can improve your application's performance.

What dynamic rendering is and when to use it.

Different approaches to make your dashboard dynamic.

Simulate a slow data fetch to see what happens.

### [What is Static Rendering?](https://nextjs.org/learn/dashboard-app/static-and-dynamic-rendering#what-is-static-rendering)

With static rendering, data fetching and rendering happens on the server at build time (when you deploy) or during [revalidation](https://nextjs.org/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating#revalidating-data). The result can then be distributed and cached in a [Content Delivery Network (CDN)](https://nextjs.org/docs/app/building-your-application/rendering/server-components#static-rendering-default).

![Diagram showing how users hit the CDN instead of the server when requesting a page](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fstatic-site-generation.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

Whenever a user visits your application, the cached result is served. There are a couple of benefits of static rendering:

- \*\*Faster Websites\*\* - Prerendered content can be cached and globally distributed. This ensures that users around the world can access your website's content more quickly and reliably.

- \*\*Reduced Server Load\*\* - Because the content is cached, your server does not have to dynamically generate content for each user request.

- \*\*SEO\*\* - Prerendered content is easier for search engine crawlers to index, as the content is already available when the page loads. This can lead to improved search engine rankings.

Static rendering is useful for UI with \*\*no data\*\* or \*\*data that is shared across users\*\*, such as a static blog post or a product page. It might not be a good fit for a dashboard that has personalized data that is regularly updated.

The opposite of static rendering is dynamic rendering.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

Why might static rendering not be a good fit for a dashboard app?

## [What is Dynamic Rendering?](https://nextjs.org/learn/dashboard-app/static-and-dynamic-rendering#what-is-dynamic-rendering)

With dynamic rendering, content is rendered on the server for each user at \*\*request time\*\* (when the user visits the page). There are a couple of benefits of dynamic rendering:

- \*\*Real-Time Data\*\* - Dynamic rendering allows your application to display real-time or frequently updated data. This is ideal for applications where data changes often.

- \*\*User-Specific Content\*\* - It's easier to serve personalized content, such as dashboards or user profiles, and update the data based on user interaction.

- \*\*Request Time Information\*\* - Dynamic rendering allows you to access information that can only be known at request time, such as cookies or the URL search parameters.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

What kind of information can only be known at request time?

## [Making the dashboard dynamic](https://nextjs.org/learn/dashboard-app/static-and-dynamic-rendering#making-the-dashboard-dynamic)

By default, `@vercel/postgres` doesn't set its own caching semantics. This allows the framework to set its own static and dynamic behavior.

You can use a Next.js API called `unstable\_noStore` inside your Server Components or data fetching functions to opt out of static rendering. Let's add this.

In your `data.ts`, import `unstable\_noStore` from `next/cache`, and call it the top of your data fetching functions:

/app/lib/data.ts

```

// ...import { unstable\_noStore as noStore } from 'next/cache'; export async function fetchRevenue() { // Add noStore() here to prevent the response from being cached. // This is equivalent to in fetch(..., {cache: 'no-store'}). noStore(); // ...} export async function fetchLatestInvoices() { noStore(); // ...} export async function fetchCardData() { noStore(); // ...} export async function fetchFilteredInvoices( query: string, currentPage: number,) { noStore(); // ...} export async function fetchInvoicesPages(query: string) { noStore(); // ...} export async function fetchFilteredCustomers(query: string) { noStore(); // ...} export async function fetchInvoiceById(query: string) { noStore(); // ...}

```

> \*\*Note:\*\* `unstable\_noStore` is an experimental API and may change in the future. If you prefer to use a stable API in your own projects, you can also use the [Segment Config Option](https://nextjs.org/docs/app/api-reference/file-conventions/route-segment-config) `export const dynamic = "force-dynamic"`.

## [Simulating a Slow Data Fetch](https://nextjs.org/learn/dashboard-app/static-and-dynamic-rendering#simulating-a-slow-data-fetch)

Making the dashboard dynamic is a good first step. However... there is still one problem we mentioned in the previous chapter. What happens if one data request is slower than all the others?

Let's simulate a slow data fetch. In your `data.ts` file, uncomment the `console.log` and `setTimeout` inside `fetchRevenue()`:

/app/lib/data.ts

```

export async function fetchRevenue() { try { // We artificially delay a response for demo purposes. // Don't do this in production :) console.log('Fetching revenue data...'); await new Promise((resolve) => setTimeout(resolve, 3000)); const data = await sql<Revenue>`SELECT \* FROM revenue`; console.log('Data fetch completed after 3 seconds.'); return data.rows; } catch (error) { console.error('Database Error:', error); throw new Error('Failed to fetch revenue data.'); }}

```

Now open [http://localhost:3000/dashboard/](http://localhost:3000/dashboard/) in a new tab and notice how the page takes longer to load. In your terminal, you should also see the following messages:

```

Fetching revenue data...Data fetch completed after 3 seconds.

```

Here, you've added an artificial 3-second delay to simulate a slow data fetch. The result is that now your whole page is blocked while the data is being fetched.

Which brings us to a common challenge developers have to solve:

With dynamic rendering, \*\*your application is only as fast as your slowest data fetch.\*\*

8

## You've Completed Chapter 8

Nice! You've just learned about static and dynamic rendering in Next.js.

# Streaming

In the previous chapter, you made your dashboard page dynamic, however, we discussed how the slow data fetches can impact the performance of your application. Let's look at how you can improve the user experience when there are slow data requests.

In this chapter...

Here are the topics we’ll cover

What streaming is and when you might use it.

How to implement streaming with `loading.tsx` and Suspense.

What loading skeletons are.

What route groups are, and when you might use them.

Where to place Suspense boundaries in your application.

## [What is streaming?](https://nextjs.org/learn/dashboard-app/streaming#what-is-streaming)

Streaming is a data transfer technique that allows you to break down a route into smaller "chunks" and progressively stream them from the server to the client as they become ready.

![Diagram showing time with sequential data fetching and parallel data fetching](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fserver-rendering-with-streaming.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

By streaming, you can prevent slow data requests from blocking your whole page. This allows the user to see and interact with parts of the page without waiting for all the data to load before any UI can be shown to the user.

![Diagram showing time with sequential data fetching and parallel data fetching](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fserver-rendering-with-streaming-chart.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

Streaming works well with React's component model, as each component can be considered a \_chunk\_.

There are two ways you implement streaming in Next.js:

1. At the page level, with the `loading.tsx` file.

2. For specific components, with `<Suspense>`.

Let's see how this works.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

What is one advantage of streaming?

## [Streaming a whole page with `loading.tsx`](https://nextjs.org/learn/dashboard-app/streaming#streaming-a-whole-page-with-loadingtsx)

In the `/app/dashboard` folder, create a new file called `loading.tsx`:

/app/dashboard/loading.tsx

```

export default function Loading() { return <div>Loading...</div>;}

```

Refresh [http://localhost:3000/dashboard](http://localhost:3000/dashboard), and you should now see:

![Dashboard page with 'Loading...' text](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Floading-page.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

A few things are happening here:

1. `loading.tsx` is a special Next.js file built on top of Suspense, it allows you to create fallback UI to show as a replacement while page content loads.

2. Since `<Sidebar>` is static, so it's shown immediately. The user can interact with `<Sidebar>` while the dynamic content is loading.

3. The user doesn't have to wait for the page to finish loading before navigating away (this is called interruptable navigation).

Congratulations! You've just implemented streaming. But we can do more to improve the user experience. Let's show a loading skeleton instead of the `Loading…` text.

### [Adding loading skeletons](https://nextjs.org/learn/dashboard-app/streaming#adding-loading-skeletons)

A loading skeleton is a simplified version of the UI. Many websites use them as a placeholder (or fallback) to indicate to users that the content is loading. Any UI you embed into `loading.tsx` will be embedded as part of the static file, and sent first. Then, the rest of the dynamic content will be streamed from the server to the client.

Inside your `loading.tsx` file, import a new component called `<DashboardSkeleton>`:

/app/dashboard/loading.tsx

```

import DashboardSkeleton from '@/app/ui/skeletons'; export default function Loading() { return <DashboardSkeleton />;}

```

Then, refresh [http://localhost:3000/dashboard](http://localhost:3000/dashboard), and you should now see:

![Dashboard page with loading skeletons](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Floading-page-with-skeleton.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

### [Fixing the loading skeleton bug with route groups](https://nextjs.org/learn/dashboard-app/streaming#fixing-the-loading-skeleton-bug-with-route-groups)

Right now, your loading skeleton will apply to the invoices and customers pages as well.

Since `loading.tsx` is a level higher than `/invoices/page.tsx` and `/customers/page.tsx` in the file system, it's also applied to those pages.

We can change this with [Route Groups](https://nextjs.org/docs/app/building-your-application/routing/route-groups). Create a new folder called `/(overview)` inside the dashboard folder. Then, move your `loading.tsx` and `page.tsx` files inside the folder:

![Folder structure showing how to create a route group using parentheses](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Froute-group.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

Now, the `loading.tsx` file will only apply to your dashboard overview page.

Route groups allow you to organize files into logical groups without affecting the URL path structure. When you create a new folder using parentheses `()`, the name won't be included in the URL path. So `/dashboard/(overview)/page.tsx` becomes `/dashboard`.

Here, you're using a route group to ensure `loading.tsx` only applies to your dashboard overview page. However, you can also use route groups to separate your application into sections (e.g. `(marketing)` routes and `(shop)` routes) or by teams for larger applications.

### [Streaming a component](https://nextjs.org/learn/dashboard-app/streaming#streaming-a-component)

So far, you're streaming a whole page. But, instead, you can be more granular and stream specific components using React Suspense.

Suspense allows you to defer rendering parts of your application until some condition is met (e.g. data is loaded). You can wrap your dynamic components in Suspense. Then, pass it a fallback component to show while the dynamic component loads.

If you remember the slow data request, `fetchRevenue()`, this is the request that is slowing down the whole page. Instead of blocking your page, you can use Suspense to stream only this component and immediately show the rest of the page's UI.

To do so, you'll need to move the data fetch to the component, let's update the code to see what that'll look like:

Delete all instances of `fetchRevenue()` and its data from `/dashboard/(overview)/page.tsx`:

/app/dashboard/(overview)/page.tsx

```

import { Card } from '@/app/ui/dashboard/cards';import RevenueChart from '@/app/ui/dashboard/revenue-chart';import LatestInvoices from '@/app/ui/dashboard/latest-invoices';import { lusitana } from '@/app/ui/fonts';import { fetchLatestInvoices, fetchCardData } from '@/app/lib/data'; // remove fetchRevenue export default async function Page() { const revenue = await fetchRevenue // delete this line const latestInvoices = await fetchLatestInvoices(); const { numberOfInvoices, numberOfCustomers, totalPaidInvoices, totalPendingInvoices, } = await fetchCardData(); return ( // ... );}

```

Then, import `<Suspense>` from React, and wrap it around `<RevenueChart />`. You can pass it a fallback component called `<RevenueChartSkeleton>`.

/app/dashboard/(overview)/page.tsx

```

import { Card } from '@/app/ui/dashboard/cards';import RevenueChart from '@/app/ui/dashboard/revenue-chart';import LatestInvoices from '@/app/ui/dashboard/latest-invoices';import { lusitana } from '@/app/ui/fonts';import { fetchLatestInvoices, fetchCardData } from '@/app/lib/data';import { Suspense } from 'react';import { RevenueChartSkeleton } from '@/app/ui/skeletons'; export default async function Page() { const latestInvoices = await fetchLatestInvoices(); const { numberOfInvoices, numberOfCustomers, totalPaidInvoices, totalPendingInvoices, } = await fetchCardData(); return ( <main> <h1 className={`${lusitana.className} mb-4 text-xl md:text-2xl`}> Dashboard </h1> <div className="grid gap-6 sm:grid-cols-2 lg:grid-cols-4"> <Card title="Collected" value={totalPaidInvoices} type="collected" /> <Card title="Pending" value={totalPendingInvoices} type="pending" /> <Card title="Total Invoices" value={numberOfInvoices} type="invoices" /> <Card title="Total Customers" value={numberOfCustomers} type="customers" /> </div> <div className="mt-6 grid grid-cols-1 gap-6 md:grid-cols-4 lg:grid-cols-8"> <Suspense fallback={<RevenueChartSkeleton />}> <RevenueChart /> </Suspense> <LatestInvoices latestInvoices={latestInvoices} /> </div> </main> );}

```

Finally, update the `<RevenueChart>` component to fetch its own data and remove the prop passed to it:

/app/ui/dashboard/revenue-chart.tsx

```

import { generateYAxis } from '@/app/lib/utils';import { CalendarIcon } from '@heroicons/react/24/outline';import { lusitana } from '@/app/ui/fonts';import { fetchRevenue } from '@/app/lib/data'; // ... export default async function RevenueChart() { // Make component async, remove the props const revenue = await fetchRevenue(); // Fetch data inside the component const chartHeight = 350; const { yAxisLabels, topLabel } = generateYAxis(revenue); if (!revenue || revenue.length === 0) { return <p className="mt-4 text-gray-400">No data available.</p>; } return ( // ... );}

```

Now refresh the page, you should see the dashboard information almost immediately, while a fallback skeleton is shown for `<RevenueChart>`:

![Dashboard page with revenue chart skeleton and loaded Card and Latest Invoices components](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Floading-revenue-chart.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

### [Practice: Streaming `<LatestInvoices>`](https://nextjs.org/learn/dashboard-app/streaming#practice-streaming-latestinvoices)

Now it's your turn! Practice what you've just learned by streaming the `<LatestInvoices>` component.

Move `fetchLatestInvoices()` down from the page to the `<LatestInvoices>` component. Wrap the component in a `<Suspense>` boundary with a fallback called `<LatestInvoicesSkeleton>`.

Once you're ready, expand the toggle to see the solution code:

Reveal the solution

## [Grouping components](https://nextjs.org/learn/dashboard-app/streaming#grouping-components)

Great! You're almost there, now you need to wrap the `<Card>` components in Suspense. You can fetch data for each individual card, but this could lead to a \_popping\_ effect as the cards load in, this can be visually jarring for the user.

So, how would you tackle this problem?

To create more of a \_staggered\_ effect, you can group the cards using a wrapper component. This means the static `<Sidebar/>` will be shown first, followed by the cards, etc.

In your `page.tsx` file:

1. Delete your `<Card>` components.

2. Delete the `fetchCardData()` function.

3. Import a new \*\*wrapper\*\* component called `<CardWrapper />`.

4. Import a new \*\*skeleton\*\* component called `<CardsSkeleton />`.

5. Wrap `<CardWrapper />` in Suspense.

/app/dashboard/page.tsx

```

import CardWrapper from '@/app/ui/dashboard/cards';// ...import { RevenueChartSkeleton, LatestInvoicesSkeleton, CardsSkeleton,} from '@/app/ui/skeletons'; export default async function Page() { return ( <main> <h1 className={`${lusitana.className} mb-4 text-xl md:text-2xl`}> Dashboard </h1> <div className="grid gap-6 sm:grid-cols-2 lg:grid-cols-4"> <Suspense fallback={<CardsSkeleton />}> <CardWrapper /> </Suspense> </div> // ... </main> );}

```

Then, move into the file `/app/ui/dashboard/cards.tsx`, import the `fetchCardData()` function, and invoke it inside the `<CardWrapper/>` component. Make sure to uncomment any necessary code in this component.

/app/ui/dashboard/cards.tsx

```

// ...import { fetchCardData } from '@/app/lib/data'; // ... export default async function CardWrapper() { const { numberOfInvoices, numberOfCustomers, totalPaidInvoices, totalPendingInvoices, } = await fetchCardData(); return ( <> <Card title="Collected" value={totalPaidInvoices} type="collected" /> <Card title="Pending" value={totalPendingInvoices} type="pending" /> <Card title="Total Invoices" value={numberOfInvoices} type="invoices" /> <Card title="Total Customers" value={numberOfCustomers} type="customers" /> </> );}

```

Refresh the page, and you should see all the cards load in at the same time. You can use this pattern when you want multiple components to load in at the same time.

## [Deciding where to place your Suspense boundaries](https://nextjs.org/learn/dashboard-app/streaming#deciding-where-to-place-your-suspense-boundaries)

Where you place your Suspense boundaries will depend on a few things:

1. How you want the user to experience the page as it streams.

2. What content you want to prioritize.

3. If the components rely on data fetching.

Take a look at your dashboard page, is there anything you would've done differently?

Don't worry. There isn't a right answer.

- You could stream the \*\*whole page\*\* like we did with `loading.tsx`... but that may lead to a longer loading time if one of the components has a slow data fetch.

- You could stream \*\*every component\*\* individually... but that may lead to UI \_popping\_ into the screen as it becomes ready.

- You could also create a \_staggered\_ effect by streaming \*\*page sections\*\*. But you'll need to create wrapper components.

Where you place your suspense boundaries will vary depending on your application. In general, it's good practice to move your data fetches down to the components that need it, and then wrap those components in Suspense. But there is nothing wrong with streaming the sections or the whole page if that's what your application needs.

Don't be afraid to experiment with Suspense and see what works best, it's a powerful API that can help you create more delightful user experiences.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

In general, what is considered good practice when working with Suspense and data fetching?

## [Looking ahead](https://nextjs.org/learn/dashboard-app/streaming#looking-ahead)

Streaming and Server Components give us new ways to handle data fetching and loading states, ultimately with the goal of improving the end user experience.

In the next chapter, you'll learn about Partial Prerendering, a new Next.js rendering model built with streaming in mind.

## You've Completed Chapter 9

You've learned how to stream components with Suspense and loading skeletons.

Next Up

10: Partial Prerendering (Optional)

An early look into Partial Prerendering - a new experimental rendering model built with streaming.

[Start Chapter 10](https://nextjs.org/learn/dashboard-app/partial-prerendering)

10

Chapter 10

# Partial Prerendering (Optional)

> Partial Prerendering is an experimental feature introduced in Next.js 14. The content of this page may be updated as the feature progresses in stability. You may want to skip this chapter if you prefer to not use experimental features. This chapter is not required to complete the course.

In this chapter...

Here are the topics we’ll cover

What Partial Prerendering is.

How Partial Prerendering works.

## [Combining Static and Dynamic Content](https://nextjs.org/learn/dashboard-app/partial-prerendering#combining-static-and-dynamic-content)

Currently, if you call a [dynamic function](https://nextjs.org/docs/app/building-your-application/routing/route-handlers#dynamic-functions) inside your route (e.g. `noStore()`, `cookies()`, etc), your whole route becomes dynamic.

This aligns with how most web apps are built today, you either choose between static and dynamic rendering for your \*\*entire application\*\* or for \*\*specific routes\*\*.

However, most routes are not fully static or dynamic. You may have a route that has both static and dynamic content. For example, let's say you have a social media feed, the posts would be static, but the likes for the post would be dynamic. Or an ecommerce site, where the product details are static, but the user's cart is dynamic.

Going back to your dashboard page, what components would you consider static vs. dynamic?

Once you're ready, click the button below to see how we would split the dashboard route:

Reveal the solution

## [What is Partial Prerendering?](https://nextjs.org/learn/dashboard-app/partial-prerendering#what-is-partial-prerendering)

In Next.js 14, there is a preview of a new rendering model called \*\*Partial Prerendering\*\*. Partial Prerendering is an experimental feature that allows you to render a route with a static loading shell, while keeping some parts dynamic. In other words, you can isolate the dynamic parts of a route. For example:

![Partially Prerendered Product Page showing static nav and product information, and dynamic cart and recommended products](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fthinking-in-ppr.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

When a user visits a route:

- A static route \_shell\_ is served, this makes the initial load fast.

- The shell leaves \_holes\_ where dynamic content will load in async.

- The async holes are loaded in parallel, reducing the overall load time of the page.

This is different from how your application behaves today, where entire routes are either fully static or dynamic.

Partial Prerendering combines ultra-quick static edge delivery with fully dynamic capabilities and we believe it has the potential to [become the default rendering model for web applications](https://vercel.com/blog/partial-prerendering-with-next-js-creating-a-new-default-rendering-model), bringing together the best of static site generation and dynamic delivery.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

## [How does Partial Prerendering work?](https://nextjs.org/learn/dashboard-app/partial-prerendering#how-does-partial-prerendering-work)

Partial Prerendering leverages React's [Concurrent APIs](https://react.dev/blog/2021/12/17/react-conf-2021-recap#react-18-and-concurrent-features) and uses [Suspense](https://react.dev/reference/react/Suspense) to defer rendering parts of your application until some condition is met (e.g. data is loaded).

The fallback is embedded into the initial static file along with other static content. At build time (or during revalidation), the static parts of the route are \_prerendered\_, and the rest is \_postponed\_ until the user requests the route.

It's worth noting that wrapping a component in Suspense doesn't make the component itself dynamic (remember you used `unstable\_noStore` to achieve this behavior), but rather Suspense is used as a boundary between the static and dynamic parts of your route.

The great thing about Partial Prerendering is that you don't need to change your code to use it. As long as you're using Suspense to wrap the dynamic parts of your route, Next.js will know which parts of your route are static and which are dynamic.

> \*\*Note:\*\* To learn more about how Partial Prerendering can be configured, see the [Partial Prerendering (experimental) documentation](https://nextjs.org/docs/app/api-reference/next-config-js/partial-prerendering) or try the [Partial Prerendering template and demo](https://vercel.com/templates/next.js/partial-prerendering-nextjs). It's important to note that this feature is \*\*experimental\*\* and \*\*not yet ready for production deployment\*\*.

## [Summary](https://nextjs.org/learn/dashboard-app/partial-prerendering#summary)

To recap, you've done a few things to optimize data fetching in your application, you've:

1. Created a database in the same region as your application code to reduce latency between your server and database.

2. Fetched data on the server with React Server Components. This allows you to keep expensive data fetches and logic on the server, reduces the client-side JavaScript bundle, and prevents your database secrets from being exposed to the client.

3. Used SQL to only fetch the data you needed, reducing the amount of data transferred for each request and the amount of JavaScript needed to transform the data in-memory.

4. Parallelize data fetching with JavaScript - where it made sense to do so.

5. Implemented Streaming to prevent slow data requests from blocking your whole page, and to allow the user to start interacting with the UI without waiting for everything to load.

6. Move data fetching down to the components that need it, thus isolating which parts of your routes should be dynamic in preparation for Partial Prerendering.

In the next chapter, we'll look at two common patterns you might need to implement when fetching data: search and pagination.

# Adding Search and Pagination

In the previous chapter, you improved your dashboard's initial loading performance with streaming. Now let's move on to the `/invoices` page, and learn how to add search and pagination!

In this chapter...

Here are the topics we’ll cover

Learn how to use the Next.js APIs: `searchParams`, `usePathname`, and `useRouter`.

Implement search and pagination using URL search params.

## [Starting code](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#starting-code)

Inside your `/dashboard/invoices/page.tsx` file, paste the following code:

/app/dashboard/invoices/page.tsx

```

import Pagination from '@/app/ui/invoices/pagination';import Search from '@/app/ui/search';import Table from '@/app/ui/invoices/table';import { CreateInvoice } from '@/app/ui/invoices/buttons';import { lusitana } from '@/app/ui/fonts';import { InvoicesTableSkeleton } from '@/app/ui/skeletons';import { Suspense } from 'react'; export default async function Page() { return ( <div className="w-full"> <div className="flex w-full items-center justify-between"> <h1 className={`${lusitana.className} text-2xl`}>Invoices</h1> </div> <div className="mt-4 flex items-center justify-between gap-2 md:mt-8"> <Search placeholder="Search invoices..." /> <CreateInvoice /> </div> {/\* <Suspense key={query + currentPage} fallback={<InvoicesTableSkeleton />}> <Table query={query} currentPage={currentPage} /> </Suspense> \*/} <div className="mt-5 flex w-full justify-center"> {/\* <Pagination totalPages={totalPages} /> \*/} </div> </div> );}

```

Spend some time familiarizing yourself with the page and the components you'll be working with:

1. `<Search/>` allows users to search for specific invoices.

2. `<Pagination/>` allows users to navigate between pages of invoices.

3. `<Table/>` displays the invoices.

Your search functionality will span the client and the server. When a user searches for an invoice on the client, the URL params will be updated, data will be fetched on the server, and the table will re-render on the server with the new data.

## [Why use URL search params?](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#why-use-url-search-params)

As mentioned above, you'll be using URL search params to manage the search state. This pattern may be new if you're used to doing it with client side state.

There are a couple of benefits of implementing search with URL params:

- \*\*Bookmarkable and Shareable URLs\*\*: Since the search parameters are in the URL, users can bookmark the current state of the application, including their search queries and filters, for future reference or sharing.

- \*\*Server-Side Rendering and Initial Load\*\*: URL parameters can be directly consumed on the server to render the initial state, making it easier to handle server rendering.

- \*\*Analytics and Tracking\*\*: Having search queries and filters directly in the URL makes it easier to track user behavior without requiring additional client-side logic.

## [Adding the search functionality](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#adding-the-search-functionality)

These are the Next.js client hooks that you'll use to implement the search functionality:

- \*\*`useSearchParams`\*\*- Allows you to access the parameters of the current URL. For example, the search params for this URL `/dashboard/invoices?page=1&query=pending` would look like this: `{page: '1', query: 'pending'}`.

- \*\*`usePathname`\*\* - Lets you read the current URL's pathname. For example, for the route `/dashboard/invoices`, `usePathname` would return `'/dashboard/invoices'`.

- \*\*`useRouter`\*\* - Enables navigation between routes within client components programmatically. There are [multiple methods](https://nextjs.org/docs/app/api-reference/functions/use-router#userouter) you can use.

Here's a quick overview of the implementation steps:

1. Capture the user's input.

2. Update the URL with the search params.

3. Keep the URL in sync with the input field.

4. Update the table to reflect the search query.

### [1. Capture the user's input](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#1-capture-the-users-input)

Go into the `<Search>` Component (`/app/ui/search.tsx`), and you'll notice:

- `"use client"` - This is a Client Component, which means you can use event listeners and hooks.

- `<input>` - This is the search input.

Create a new `handleSearch` function, and add an `onChange` listener to the `<input>` element. `onChange` will invoke `handleSearch` whenever the input value changes.

/app/ui/search.tsx

```

'use client'; import { MagnifyingGlassIcon } from '@heroicons/react/24/outline'; export default function Search({ placeholder }: { placeholder: string }) { function handleSearch(term: string) { console.log(term); } return ( <div className="relative flex flex-1 flex-shrink-0"> <label htmlFor="search" className="sr-only"> Search </label> <input className="peer block w-full rounded-md border border-gray-200 py-[9px] pl-10 text-sm outline-2 placeholder:text-gray-500" placeholder={placeholder} onChange={(e) => { handleSearch(e.target.value); }} /> <MagnifyingGlassIcon className="absolute left-3 top-1/2 h-[18px] w-[18px] -translate-y-1/2 text-gray-500 peer-focus:text-gray-900" /> </div> );}

```

Test that it's working correctly by opening the console in your Developer Tools, then type into the search field. You should see the search term logged to the console.

Great! You're capturing the user's search input. Now, you need to update the URL with the search term.

### [2. Update the URL with the search params](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#2-update-the-url-with-the-search-params)

Import the `useSearchParams` hook from `'next/navigation'`, and assign it to a variable:

/app/ui/search.tsx

```

'use client'; import { MagnifyingGlassIcon } from '@heroicons/react/24/outline';import { useSearchParams } from 'next/navigation'; export default function Search() { const searchParams = useSearchParams(); function handleSearch(term: string) { console.log(term); } // ...}

```

Inside `handleSearch,` create a new [`URLSearchParams`](https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams) instance using your new `searchParams` variable.

/app/ui/search.tsx

```

'use client'; import { MagnifyingGlassIcon } from '@heroicons/react/24/outline';import { useSearchParams } from 'next/navigation'; export default function Search() { const searchParams = useSearchParams(); function handleSearch(term: string) { const params = new URLSearchParams(searchParams); } // ...}

```

`URLSearchParams` is a Web API that provides utility methods for manipulating the URL query parameters. Instead of creating a complex string literal, you can use it to get the params string like `?page=1&query=a`.

Next, `set` the params string based on the user’s input. If the input is empty, you want to `delete` it:

/app/ui/search.tsx

```

'use client'; import { MagnifyingGlassIcon } from '@heroicons/react/24/outline';import { useSearchParams } from 'next/navigation'; export default function Search() { const searchParams = useSearchParams(); function handleSearch(term: string) { const params = new URLSearchParams(searchParams); if (term) { params.set('query', term); } else { params.delete('query'); } } // ...}

```

Now that you have the query string. You can use Next.js's `useRouter` and `usePathname` hooks to update the URL.

Import `useRouter` and `usePathname` from `'next/navigation'`, and use the `replace` method from `useRouter()` inside `handleSearch`:

/app/ui/search.tsx

```

'use client'; import { MagnifyingGlassIcon } from '@heroicons/react/24/outline';import { useSearchParams, usePathname, useRouter } from 'next/navigation'; export default function Search() { const searchParams = useSearchParams(); const pathname = usePathname(); const { replace } = useRouter(); function handleSearch(term: string) { const params = new URLSearchParams(searchParams); if (term) { params.set('query', term); } else { params.delete('query'); } replace(`${pathname}?${params.toString()}`); }}

```

Here's a breakdown of what's happening:

- `${pathname}` is the current path, in your case, `"/dashboard/invoices"`.

- As the user types into the search bar, `params.toString()` translates this input into a URL-friendly format.

- `replace(${pathname}?${params.toString()})` updates the URL with the user's search data. For example, `/dashboard/invoices?query=lee` if the user searches for "Lee".

- The URL is updated without reloading the page, thanks to Next.js's client-side navigation (which you learned about in the chapter on [navigating between pages](https://nextjs.org/learn/dashboard-app/navigating-between-pages).

### [3. Keeping the URL and input in sync](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#3-keeping-the-url-and-input-in-sync)

To ensure the input field is in sync with the URL and will be populated when sharing, you can pass a `defaultValue` to input by reading from `searchParams`:

/app/ui/search.tsx

```

<input className="peer block w-full rounded-md border border-gray-200 py-[9px] pl-10 text-sm outline-2 placeholder:text-gray-500" placeholder={placeholder} onChange={(e) => { handleSearch(e.target.value); }} defaultValue={searchParams.get('query')?.toString()}/>

```

> \*\*`defaultValue` vs. `value` / Controlled vs. Uncontrolled\*\*

>

> If you're using state to manage the value of an input, you'd use the `value` attribute to make it a controlled component. This means React would manage the input's state.

>

> However, since you're not using state, you can use `defaultValue`. This means the native input will manage its own state. This is okay since you're saving the search query to the URL instead of state.

### [4. Updating the table](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#4-updating-the-table)

Finally, you need to update the table component to reflect the search query.

Navigate back to the invoices page.

Page components [accept a prop called `searchParams`](https://nextjs.org/docs/app/api-reference/file-conventions/page), so you can pass the current URL params to the `<Table>` component.

/app/dashboard/invoices/page.tsx

```

import Pagination from '@/app/ui/invoices/pagination';import Search from '@/app/ui/search';import Table from '@/app/ui/invoices/table';import { CreateInvoice } from '@/app/ui/invoices/buttons';import { lusitana } from '@/app/ui/fonts';import { Suspense } from 'react';import { InvoicesTableSkeleton } from '@/app/ui/skeletons'; export default async function Page({ searchParams,}: { searchParams?: { query?: string; page?: string; };}) { const query = searchParams?.query || ''; const currentPage = Number(searchParams?.page) || 1; return ( <div className="w-full"> <div className="flex w-full items-center justify-between"> <h1 className={`${lusitana.className} text-2xl`}>Invoices</h1> </div> <div className="mt-4 flex items-center justify-between gap-2 md:mt-8"> <Search placeholder="Search invoices..." /> <CreateInvoice /> </div> <Suspense key={query + currentPage} fallback={<InvoicesTableSkeleton />}> <Table query={query} currentPage={currentPage} /> </Suspense> <div className="mt-5 flex w-full justify-center"> {/\* <Pagination totalPages={totalPages} /> \*/} </div> </div> );}

```

If you navigate to the `<Table>` Component, you'll see that the two props, `query` and `currentPage`, are passed to the `fetchFilteredInvoices()` function which returns the invoices that match the query.

/app/ui/invoices/table.tsx

```

// ...export default async function InvoicesTable({ query, currentPage,}: { query: string; currentPage: number;}) { const invoices = await fetchFilteredInvoices(query, currentPage); // ...}

```

With these changes in place, go ahead and test it out. If you search for a term, you'll update the URL, which will send a new request to the server, data will be fetched on the server, and only the invoices that match your query will be returned.

> \*\*When to use the `useSearchParams()` hook vs. the `searchParams` prop?\*\*

>

> You might have noticed you used two different ways to extract search params. Whether you use one or the other depends on whether you're working on the client or the server.

>

> - `<Search>` is a Client Component, so you used the `useSearchParams()` hook to access the params from the client.

> - `<Table>` is a Server Component that fetches its own data, so you can pass the `searchParams` prop from the page to the component.

>

> As a general rule, if you want to read the params from the client, use the `useSearchParams()` hook as this avoids having to go back to the server.

### [Best practice: Debouncing](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#best-practice-debouncing)

Congratulations! You've implemented search with Next.js! But there's something you can do to optimize it.

Inside your `handleSearch` function, add the following `console.log`:

/app/ui/search.tsx

```

function handleSearch(term: string) { console.log(`Searching... ${term}`); const params = new URLSearchParams(searchParams); if (term) { params.set('query', term); } else { params.delete('query'); } replace(`${pathname}?${params.toString()}`);}

```

Then type "Emil" into your search bar and check the console in dev tools. What is happening?

Dev Tools Console

```

Searching... ESearching... EmSearching... EmiSearching... Emil

```

You're updating the URL on every keystroke, and therefore querying your database on every keystroke! This isn't a problem as our application is small, but imagine if your application had thousands of users, each sending a new request to your database on each keystroke.

\*\*Debouncing\*\* is a programming practice that limits the rate at which a function can fire. In our case, you only want to query the database when the user has stopped typing.

> \*\*How Debouncing Works:\*\*

>

> 1. \*\*Trigger Event\*\*: When an event that should be debounced (like a keystroke in the search box) occurs, a timer starts.

> 2. \*\*Wait\*\*: If a new event occurs before the timer expires, the timer is reset.

> 3. \*\*Execution\*\*: If the timer reaches the end of its countdown, the debounced function is executed.

You can implement debouncing in a few ways, including manually creating your own debounce function. To keep things simple, we'll use a library called [`use-debounce`](https://www.npmjs.com/package/use-debounce).

Install `use-debounce`:

Terminal

```

npm i use-debounce

```

In your `<Search>` Component, import a function called `useDebouncedCallback`:

/app/ui/search.tsx

```

// ...import { useDebouncedCallback } from 'use-debounce'; // Inside the Search Component...const handleSearch = useDebouncedCallback((term) => { console.log(`Searching... ${term}`); const params = new URLSearchParams(searchParams); if (term) { params.set('query', term); } else { params.delete('query'); } replace(`${pathname}?${params.toString()}`);}, 300);

```

This function will wrap the contents of `handleSearch`, and only run the code after a specific time once the user has stopped typing (300ms).

Now type in your search bar again, and open the console in dev tools. You should see the following:

Dev Tools Console

```

Searching... Emil

```

By debouncing, you can reduce the number of requests sent to your database, thus saving resources.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

## [Adding pagination](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#adding-pagination)

After introducing the search feature, you'll notice the table displays only 6 invoices at a time. This is because the `fetchFilteredInvoices()` function in `data.ts` returns a maximum of 6 invoices per page.

Adding pagination allows users to navigate through the different pages to view all the invoices. Let's see how you can implement pagination using URL params, just like you did with search.

Navigate to the `<Pagination/>` component and you'll notice that it's a Client Component. You don't want to fetch data on the client as this would expose your database secrets (remember, you're not using an API layer). Instead, you can fetch the data on the server, and pass it to the component as a prop.

In `/dashboard/invoices/page.tsx`, import a new function called `fetchInvoicesPages` and pass the `query` from `searchParams` as an argument:

/app/dashboard/invoices/page.tsx

```

// ...import { fetchInvoicesPages } from '@/app/lib/data'; export default async function Page({ searchParams,}: { searchParams?: { query?: string, page?: string, },}) { const query = searchParams?.query || ''; const currentPage = Number(searchParams?.page) || 1; const totalPages = await fetchInvoicesPages(query); return ( // ... );}

```

`fetchInvoicesPages` returns the total number of pages based on the search query. For example, if there are 12 invoices that match the search query, and each page displays 6 invoices, then the total number of pages would be 2.

Next, pass the `totalPages` prop to the `<Pagination/>` component:

/app/dashboard/invoices/page.tsx

```

// ... export default async function Page({ searchParams,}: { searchParams?: { query?: string; page?: string; };}) { const query = searchParams?.query || ''; const currentPage = Number(searchParams?.page) || 1; const totalPages = await fetchInvoicesPages(query); return ( <div className="w-full"> <div className="flex w-full items-center justify-between"> <h1 className={`${lusitana.className} text-2xl`}>Invoices</h1> </div> <div className="mt-4 flex items-center justify-between gap-2 md:mt-8"> <Search placeholder="Search invoices..." /> <CreateInvoice /> </div> <Suspense key={query + currentPage} fallback={<InvoicesTableSkeleton />}> <Table query={query} currentPage={currentPage} /> </Suspense> <div className="mt-5 flex w-full justify-center"> <Pagination totalPages={totalPages} /> </div> </div> );}

```

Navigate to the `<Pagination/>` component and import the `usePathname` and `useSearchParams` hooks. We will use this to get the current page and set the new page. Make sure to also uncomment the code in this component. Your application will break temporarily as you haven't implemented the `<Pagination/>` logic yet. Let's do that now!

/app/ui/invoices/pagination.tsx

```

'use client'; import { ArrowLeftIcon, ArrowRightIcon } from '@heroicons/react/24/outline';import clsx from 'clsx';import Link from 'next/link';import { generatePagination } from '@/app/lib/utils';import { usePathname, useSearchParams } from 'next/navigation'; export default function Pagination({ totalPages }: { totalPages: number }) { const pathname = usePathname(); const searchParams = useSearchParams(); const currentPage = Number(searchParams.get('page')) || 1; // ...}

```

Next, create a new function inside the `<Pagination>` Component called `createPageURL`. Similarly to the search, you'll use `URLSearchParams` to set the new page number, and `pathName` to create the URL string.

/app/ui/invoices/pagination.tsx

```

'use client'; import { ArrowLeftIcon, ArrowRightIcon } from '@heroicons/react/24/outline';import clsx from 'clsx';import Link from 'next/link';import { generatePagination } from '@/app/lib/utils';import { usePathname, useSearchParams } from 'next/navigation'; export default function Pagination({ totalPages }: { totalPages: number }) { const pathname = usePathname(); const searchParams = useSearchParams(); const currentPage = Number(searchParams.get('page')) || 1; const createPageURL = (pageNumber: number | string) => { const params = new URLSearchParams(searchParams); params.set('page', pageNumber.toString()); return `${pathname}?${params.toString()}`; }; // ...}

```

Here's a breakdown of what's happening:

- `createPageURL` creates an instance of the current search parameters.

- Then, it updates the "page" parameter to the provided page number.

- Finally, it constructs the full URL using the pathname and updated search parameters.

The rest of the `<Pagination>` component deals with styling and different states (first, last, active, disabled, etc). We won't go into detail for this course, but feel free to look through the code to see where `createPageURL` is being called.

Finally, when the user types a new search query, you want to reset the page number to 1. You can do this by updating the `handleSearch` function in your `<Search>` component:

/app/ui/search.tsx

```

'use client'; import { MagnifyingGlassIcon } from '@heroicons/react/24/outline';import { usePathname, useRouter, useSearchParams } from 'next/navigation';import { useDebouncedCallback } from 'use-debounce'; export default function Search({ placeholder }: { placeholder: string }) { const searchParams = useSearchParams(); const { replace } = useRouter(); const pathname = usePathname(); const handleSearch = useDebouncedCallback((term) => { const params = new URLSearchParams(searchParams); params.set('page', '1'); if (term) { params.set('query', term); } else { params.delete('query'); } replace(`${pathname}?${params.toString()}`); }, 300);

```

## [Summary](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#summary)

Congratulations! You've just implemented search and pagination using URL Params and Next.js APIs.

To summarize, in this chapter:

- You've handled search and pagination with URL search parameters instead of client state.

- You've fetched data on the server.

- You're using the `useRouter` router hook for smoother, client-side transitions.

These patterns are different from what you may be used to when working with client-side React, but hopefully, you now better understand the benefits of using URL search params and lifting this state to the server.

## You've Completed Chapter 11

Your dashboard now has search and pagination functionality!

# Adding Search and Pagination

In the previous chapter, you improved your dashboard's initial loading performance with streaming. Now let's move on to the `/invoices` page, and learn how to add search and pagination!

In this chapter...

Here are the topics we’ll cover

Learn how to use the Next.js APIs: `searchParams`, `usePathname`, and `useRouter`.

Implement search and pagination using URL search params.

## [Starting code](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#starting-code)

Inside your `/dashboard/invoices/page.tsx` file, paste the following code:

/app/dashboard/invoices/page.tsx

```

import Pagination from '@/app/ui/invoices/pagination';import Search from '@/app/ui/search';import Table from '@/app/ui/invoices/table';import { CreateInvoice } from '@/app/ui/invoices/buttons';import { lusitana } from '@/app/ui/fonts';import { InvoicesTableSkeleton } from '@/app/ui/skeletons';import { Suspense } from 'react'; export default async function Page() { return ( <div className="w-full"> <div className="flex w-full items-center justify-between"> <h1 className={`${lusitana.className} text-2xl`}>Invoices</h1> </div> <div className="mt-4 flex items-center justify-between gap-2 md:mt-8"> <Search placeholder="Search invoices..." /> <CreateInvoice /> </div> {/\* <Suspense key={query + currentPage} fallback={<InvoicesTableSkeleton />}> <Table query={query} currentPage={currentPage} /> </Suspense> \*/} <div className="mt-5 flex w-full justify-center"> {/\* <Pagination totalPages={totalPages} /> \*/} </div> </div> );}

```

Spend some time familiarizing yourself with the page and the components you'll be working with:

1. `<Search/>` allows users to search for specific invoices.

2. `<Pagination/>` allows users to navigate between pages of invoices.

3. `<Table/>` displays the invoices.

Your search functionality will span the client and the server. When a user searches for an invoice on the client, the URL params will be updated, data will be fetched on the server, and the table will re-render on the server with the new data.

## [Why use URL search params?](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#why-use-url-search-params)

As mentioned above, you'll be using URL search params to manage the search state. This pattern may be new if you're used to doing it with client side state.

There are a couple of benefits of implementing search with URL params:

- \*\*Bookmarkable and Shareable URLs\*\*: Since the search parameters are in the URL, users can bookmark the current state of the application, including their search queries and filters, for future reference or sharing.

- \*\*Server-Side Rendering and Initial Load\*\*: URL parameters can be directly consumed on the server to render the initial state, making it easier to handle server rendering.

- \*\*Analytics and Tracking\*\*: Having search queries and filters directly in the URL makes it easier to track user behavior without requiring additional client-side logic.

## [Adding the search functionality](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#adding-the-search-functionality)

These are the Next.js client hooks that you'll use to implement the search functionality:

- \*\*`useSearchParams`\*\*- Allows you to access the parameters of the current URL. For example, the search params for this URL `/dashboard/invoices?page=1&query=pending` would look like this: `{page: '1', query: 'pending'}`.

- \*\*`usePathname`\*\* - Lets you read the current URL's pathname. For example, for the route `/dashboard/invoices`, `usePathname` would return `'/dashboard/invoices'`.

- \*\*`useRouter`\*\* - Enables navigation between routes within client components programmatically. There are [multiple methods](https://nextjs.org/docs/app/api-reference/functions/use-router#userouter) you can use.

Here's a quick overview of the implementation steps:

1. Capture the user's input.

2. Update the URL with the search params.

3. Keep the URL in sync with the input field.

4. Update the table to reflect the search query.

### [1. Capture the user's input](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#1-capture-the-users-input)

Go into the `<Search>` Component (`/app/ui/search.tsx`), and you'll notice:

- `"use client"` - This is a Client Component, which means you can use event listeners and hooks.

- `<input>` - This is the search input.

Create a new `handleSearch` function, and add an `onChange` listener to the `<input>` element. `onChange` will invoke `handleSearch` whenever the input value changes.

/app/ui/search.tsx

```

'use client'; import { MagnifyingGlassIcon } from '@heroicons/react/24/outline'; export default function Search({ placeholder }: { placeholder: string }) { function handleSearch(term: string) { console.log(term); } return ( <div className="relative flex flex-1 flex-shrink-0"> <label htmlFor="search" className="sr-only"> Search </label> <input className="peer block w-full rounded-md border border-gray-200 py-[9px] pl-10 text-sm outline-2 placeholder:text-gray-500" placeholder={placeholder} onChange={(e) => { handleSearch(e.target.value); }} /> <MagnifyingGlassIcon className="absolute left-3 top-1/2 h-[18px] w-[18px] -translate-y-1/2 text-gray-500 peer-focus:text-gray-900" /> </div> );}

```

Test that it's working correctly by opening the console in your Developer Tools, then type into the search field. You should see the search term logged to the console.

Great! You're capturing the user's search input. Now, you need to update the URL with the search term.

### [2. Update the URL with the search params](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#2-update-the-url-with-the-search-params)

Import the `useSearchParams` hook from `'next/navigation'`, and assign it to a variable:

/app/ui/search.tsx

```

'use client'; import { MagnifyingGlassIcon } from '@heroicons/react/24/outline';import { useSearchParams } from 'next/navigation'; export default function Search() { const searchParams = useSearchParams(); function handleSearch(term: string) { console.log(term); } // ...}

```

Inside `handleSearch,` create a new [`URLSearchParams`](https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams) instance using your new `searchParams` variable.

/app/ui/search.tsx

```

'use client'; import { MagnifyingGlassIcon } from '@heroicons/react/24/outline';import { useSearchParams } from 'next/navigation'; export default function Search() { const searchParams = useSearchParams(); function handleSearch(term: string) { const params = new URLSearchParams(searchParams); } // ...}

```

`URLSearchParams` is a Web API that provides utility methods for manipulating the URL query parameters. Instead of creating a complex string literal, you can use it to get the params string like `?page=1&query=a`.

Next, `set` the params string based on the user’s input. If the input is empty, you want to `delete` it:

/app/ui/search.tsx

```

'use client'; import { MagnifyingGlassIcon } from '@heroicons/react/24/outline';import { useSearchParams } from 'next/navigation'; export default function Search() { const searchParams = useSearchParams(); function handleSearch(term: string) { const params = new URLSearchParams(searchParams); if (term) { params.set('query', term); } else { params.delete('query'); } } // ...}

```

Now that you have the query string. You can use Next.js's `useRouter` and `usePathname` hooks to update the URL.

Import `useRouter` and `usePathname` from `'next/navigation'`, and use the `replace` method from `useRouter()` inside `handleSearch`:

/app/ui/search.tsx

```

'use client'; import { MagnifyingGlassIcon } from '@heroicons/react/24/outline';import { useSearchParams, usePathname, useRouter } from 'next/navigation'; export default function Search() { const searchParams = useSearchParams(); const pathname = usePathname(); const { replace } = useRouter(); function handleSearch(term: string) { const params = new URLSearchParams(searchParams); if (term) { params.set('query', term); } else { params.delete('query'); } replace(`${pathname}?${params.toString()}`); }}

```

Here's a breakdown of what's happening:

- `${pathname}` is the current path, in your case, `"/dashboard/invoices"`.

- As the user types into the search bar, `params.toString()` translates this input into a URL-friendly format.

- `replace(${pathname}?${params.toString()})` updates the URL with the user's search data. For example, `/dashboard/invoices?query=lee` if the user searches for "Lee".

- The URL is updated without reloading the page, thanks to Next.js's client-side navigation (which you learned about in the chapter on [navigating between pages](https://nextjs.org/learn/dashboard-app/navigating-between-pages).

### [3. Keeping the URL and input in sync](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#3-keeping-the-url-and-input-in-sync)

To ensure the input field is in sync with the URL and will be populated when sharing, you can pass a `defaultValue` to input by reading from `searchParams`:

/app/ui/search.tsx

```

<input className="peer block w-full rounded-md border border-gray-200 py-[9px] pl-10 text-sm outline-2 placeholder:text-gray-500" placeholder={placeholder} onChange={(e) => { handleSearch(e.target.value); }} defaultValue={searchParams.get('query')?.toString()}/>

```

> \*\*`defaultValue` vs. `value` / Controlled vs. Uncontrolled\*\*

>

> If you're using state to manage the value of an input, you'd use the `value` attribute to make it a controlled component. This means React would manage the input's state.

>

> However, since you're not using state, you can use `defaultValue`. This means the native input will manage its own state. This is okay since you're saving the search query to the URL instead of state.

### [4. Updating the table](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#4-updating-the-table)

Finally, you need to update the table component to reflect the search query.

Navigate back to the invoices page.

Page components [accept a prop called `searchParams`](https://nextjs.org/docs/app/api-reference/file-conventions/page), so you can pass the current URL params to the `<Table>` component.

/app/dashboard/invoices/page.tsx

```

import Pagination from '@/app/ui/invoices/pagination';import Search from '@/app/ui/search';import Table from '@/app/ui/invoices/table';import { CreateInvoice } from '@/app/ui/invoices/buttons';import { lusitana } from '@/app/ui/fonts';import { Suspense } from 'react';import { InvoicesTableSkeleton } from '@/app/ui/skeletons'; export default async function Page({ searchParams,}: { searchParams?: { query?: string; page?: string; };}) { const query = searchParams?.query || ''; const currentPage = Number(searchParams?.page) || 1; return ( <div className="w-full"> <div className="flex w-full items-center justify-between"> <h1 className={`${lusitana.className} text-2xl`}>Invoices</h1> </div> <div className="mt-4 flex items-center justify-between gap-2 md:mt-8"> <Search placeholder="Search invoices..." /> <CreateInvoice /> </div> <Suspense key={query + currentPage} fallback={<InvoicesTableSkeleton />}> <Table query={query} currentPage={currentPage} /> </Suspense> <div className="mt-5 flex w-full justify-center"> {/\* <Pagination totalPages={totalPages} /> \*/} </div> </div> );}

```

If you navigate to the `<Table>` Component, you'll see that the two props, `query` and `currentPage`, are passed to the `fetchFilteredInvoices()` function which returns the invoices that match the query.

/app/ui/invoices/table.tsx

```

// ...export default async function InvoicesTable({ query, currentPage,}: { query: string; currentPage: number;}) { const invoices = await fetchFilteredInvoices(query, currentPage); // ...}

```

With these changes in place, go ahead and test it out. If you search for a term, you'll update the URL, which will send a new request to the server, data will be fetched on the server, and only the invoices that match your query will be returned.

> \*\*When to use the `useSearchParams()` hook vs. the `searchParams` prop?\*\*

>

> You might have noticed you used two different ways to extract search params. Whether you use one or the other depends on whether you're working on the client or the server.

>

> - `<Search>` is a Client Component, so you used the `useSearchParams()` hook to access the params from the client.

> - `<Table>` is a Server Component that fetches its own data, so you can pass the `searchParams` prop from the page to the component.

>

> As a general rule, if you want to read the params from the client, use the `useSearchParams()` hook as this avoids having to go back to the server.

### [Best practice: Debouncing](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#best-practice-debouncing)

Congratulations! You've implemented search with Next.js! But there's something you can do to optimize it.

Inside your `handleSearch` function, add the following `console.log`:

/app/ui/search.tsx

```

function handleSearch(term: string) { console.log(`Searching... ${term}`); const params = new URLSearchParams(searchParams); if (term) { params.set('query', term); } else { params.delete('query'); } replace(`${pathname}?${params.toString()}`);}

```

Then type "Emil" into your search bar and check the console in dev tools. What is happening?

Dev Tools Console

```

Searching... ESearching... EmSearching... EmiSearching... Emil

```

You're updating the URL on every keystroke, and therefore querying your database on every keystroke! This isn't a problem as our application is small, but imagine if your application had thousands of users, each sending a new request to your database on each keystroke.

\*\*Debouncing\*\* is a programming practice that limits the rate at which a function can fire. In our case, you only want to query the database when the user has stopped typing.

> \*\*How Debouncing Works:\*\*

>

> 1. \*\*Trigger Event\*\*: When an event that should be debounced (like a keystroke in the search box) occurs, a timer starts.

> 2. \*\*Wait\*\*: If a new event occurs before the timer expires, the timer is reset.

> 3. \*\*Execution\*\*: If the timer reaches the end of its countdown, the debounced function is executed.

You can implement debouncing in a few ways, including manually creating your own debounce function. To keep things simple, we'll use a library called [`use-debounce`](https://www.npmjs.com/package/use-debounce).

Install `use-debounce`:

Terminal

```

npm i use-debounce

```

In your `<Search>` Component, import a function called `useDebouncedCallback`:

/app/ui/search.tsx

```

// ...import { useDebouncedCallback } from 'use-debounce'; // Inside the Search Component...const handleSearch = useDebouncedCallback((term) => { console.log(`Searching... ${term}`); const params = new URLSearchParams(searchParams); if (term) { params.set('query', term); } else { params.delete('query'); } replace(`${pathname}?${params.toString()}`);}, 300);

```

This function will wrap the contents of `handleSearch`, and only run the code after a specific time once the user has stopped typing (300ms).

Now type in your search bar again, and open the console in dev tools. You should see the following:

Dev Tools Console

```

Searching... Emil

```

By debouncing, you can reduce the number of requests sent to your database, thus saving resources.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

What problem does debouncing solve in the search feature?

## [Adding pagination](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#adding-pagination)

After introducing the search feature, you'll notice the table displays only 6 invoices at a time. This is because the `fetchFilteredInvoices()` function in `data.ts` returns a maximum of 6 invoices per page.

Adding pagination allows users to navigate through the different pages to view all the invoices. Let's see how you can implement pagination using URL params, just like you did with search.

Navigate to the `<Pagination/>` component and you'll notice that it's a Client Component. You don't want to fetch data on the client as this would expose your database secrets (remember, you're not using an API layer). Instead, you can fetch the data on the server, and pass it to the component as a prop.

In `/dashboard/invoices/page.tsx`, import a new function called `fetchInvoicesPages` and pass the `query` from `searchParams` as an argument:

/app/dashboard/invoices/page.tsx

```

// ...import { fetchInvoicesPages } from '@/app/lib/data'; export default async function Page({ searchParams,}: { searchParams?: { query?: string, page?: string, },}) { const query = searchParams?.query || ''; const currentPage = Number(searchParams?.page) || 1; const totalPages = await fetchInvoicesPages(query); return ( // ... );}

```

`fetchInvoicesPages` returns the total number of pages based on the search query. For example, if there are 12 invoices that match the search query, and each page displays 6 invoices, then the total number of pages would be 2.

Next, pass the `totalPages` prop to the `<Pagination/>` component:

/app/dashboard/invoices/page.tsx

```

// ... export default async function Page({ searchParams,}: { searchParams?: { query?: string; page?: string; };}) { const query = searchParams?.query || ''; const currentPage = Number(searchParams?.page) || 1; const totalPages = await fetchInvoicesPages(query); return ( <div className="w-full"> <div className="flex w-full items-center justify-between"> <h1 className={`${lusitana.className} text-2xl`}>Invoices</h1> </div> <div className="mt-4 flex items-center justify-between gap-2 md:mt-8"> <Search placeholder="Search invoices..." /> <CreateInvoice /> </div> <Suspense key={query + currentPage} fallback={<InvoicesTableSkeleton />}> <Table query={query} currentPage={currentPage} /> </Suspense> <div className="mt-5 flex w-full justify-center"> <Pagination totalPages={totalPages} /> </div> </div> );}

```

Navigate to the `<Pagination/>` component and import the `usePathname` and `useSearchParams` hooks. We will use this to get the current page and set the new page. Make sure to also uncomment the code in this component. Your application will break temporarily as you haven't implemented the `<Pagination/>` logic yet. Let's do that now!

/app/ui/invoices/pagination.tsx

```

'use client'; import { ArrowLeftIcon, ArrowRightIcon } from '@heroicons/react/24/outline';import clsx from 'clsx';import Link from 'next/link';import { generatePagination } from '@/app/lib/utils';import { usePathname, useSearchParams } from 'next/navigation'; export default function Pagination({ totalPages }: { totalPages: number }) { const pathname = usePathname(); const searchParams = useSearchParams(); const currentPage = Number(searchParams.get('page')) || 1; // ...}

```

Next, create a new function inside the `<Pagination>` Component called `createPageURL`. Similarly to the search, you'll use `URLSearchParams` to set the new page number, and `pathName` to create the URL string.

/app/ui/invoices/pagination.tsx

```

'use client'; import { ArrowLeftIcon, ArrowRightIcon } from '@heroicons/react/24/outline';import clsx from 'clsx';import Link from 'next/link';import { generatePagination } from '@/app/lib/utils';import { usePathname, useSearchParams } from 'next/navigation'; export default function Pagination({ totalPages }: { totalPages: number }) { const pathname = usePathname(); const searchParams = useSearchParams(); const currentPage = Number(searchParams.get('page')) || 1; const createPageURL = (pageNumber: number | string) => { const params = new URLSearchParams(searchParams); params.set('page', pageNumber.toString()); return `${pathname}?${params.toString()}`; }; // ...}

```

Here's a breakdown of what's happening:

- `createPageURL` creates an instance of the current search parameters.

- Then, it updates the "page" parameter to the provided page number.

- Finally, it constructs the full URL using the pathname and updated search parameters.

The rest of the `<Pagination>` component deals with styling and different states (first, last, active, disabled, etc). We won't go into detail for this course, but feel free to look through the code to see where `createPageURL` is being called.

Finally, when the user types a new search query, you want to reset the page number to 1. You can do this by updating the `handleSearch` function in your `<Search>` component:

/app/ui/search.tsx

```

'use client'; import { MagnifyingGlassIcon } from '@heroicons/react/24/outline';import { usePathname, useRouter, useSearchParams } from 'next/navigation';import { useDebouncedCallback } from 'use-debounce'; export default function Search({ placeholder }: { placeholder: string }) { const searchParams = useSearchParams(); const { replace } = useRouter(); const pathname = usePathname(); const handleSearch = useDebouncedCallback((term) => { const params = new URLSearchParams(searchParams); params.set('page', '1'); if (term) { params.set('query', term); } else { params.delete('query'); } replace(`${pathname}?${params.toString()}`); }, 300);

```

## [Summary](https://nextjs.org/learn/dashboard-app/adding-search-and-pagination#summary)

Congratulations! You've just implemented search and pagination using URL Params and Next.js APIs.

To summarize, in this chapter:

- You've handled search and pagination with URL search parameters instead of client state.

- You've fetched data on the server.

- You're using the `useRouter` router hook for smoother, client-side transitions.

These patterns are different from what you may be used to when working with client-side React, but hopefully, you now better understand the benefits of using URL search params and lifting this state to the server.

# Mutating Data

In the previous chapter, you implemented search and pagination using URL Search Params and Next.js APIs. Let's continue working on the Invoices page by adding the ability to create, update, and delete invoices!

In this chapter...

Here are the topics we’ll cover

What React Server Actions are and how to use them to mutate data.

How to work with forms and Server Components.

Best practices for working with the native `formData` object, including type validation.

How to revalidate the client cache using the `revalidatePath` API.

How to create dynamic route segments with specific IDs.

How to use the React’s `useFormStatus` hook for optimistic updates.

## [What are Server Actions?](https://nextjs.org/learn/dashboard-app/mutating-data#what-are-server-actions)

React Server Actions allow you to run asynchronous code directly on the server. They eliminate the need to create API endpoints to mutate your data. Instead, you write asynchronous functions that execute on the server and can be invoked from your Client or Server Components.

Security is a top priority for web applications, as they can be vulnerable to various threats. This is where Server Actions come in. They offer an effective security solution, protecting against different types of attacks, securing your data, and ensuring authorized access. Server Actions achieve this through techniques like POST requests, encrypted closures, strict input checks, error message hashing, and host restrictions, all working together to significantly enhance your app's safety.

## [Using forms with Server Actions](https://nextjs.org/learn/dashboard-app/mutating-data#using-forms-with-server-actions)

In React, you can use the `action` attribute in the `<form>` element to invoke actions. The action will automatically receive the native [FormData](https://developer.mozilla.org/en-US/docs/Web/API/FormData) object, containing the captured data.

For example:

```

// Server Componentexport default function Page() { // Action async function create(formData: FormData) { 'use server'; // Logic to mutate data... } // Invoke the action using the "action" attribute return <form action={create}>...</form>;}

```

An advantage of invoking a Server Action within a Server Component is progressive enhancement - forms work even if JavaScript is disabled on the client.

## [Next.js with Server Actions](https://nextjs.org/learn/dashboard-app/mutating-data#nextjs-with-server-actions)

Server Actions are also deeply integrated with Next.js [caching](https://nextjs.org/docs/app/building-your-application/caching). When a form is submitted through a Server Action, not only can you use the action to mutate data, but you can also revalidate the associated cache using APIs like `revalidatePath` and `revalidateTag`.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

What's one benefit of using a Server Actions?

## [Creating an invoice](https://nextjs.org/learn/dashboard-app/mutating-data#creating-an-invoice)

Here are the steps you'll take to create a new invoice:

1. Create a form to capture the user's input.

2. Create a Server Action and invoke it from the form.

3. Inside your Server Action, extract the data from the `formData` object.

4. Validate and prepare the data to be inserted into your database.

5. Insert the data and handle any errors.

6. Revalidate the cache and redirect the user back to invoices page.

### [1. Create a new route and form](https://nextjs.org/learn/dashboard-app/mutating-data#1-create-a-new-route-and-form)

To start, inside the `/invoices` folder, add a new route segment called `/create` with a `page.tsx` file:

![Invoices folder with a nested create folder, and a page.tsx file inside it](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fcreate-invoice-route.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

You'll be using this route to create new invoices. Inside your `page.tsx` file, paste the following code, then spend some time studying it:

/dashboard/invoices/create/page.tsx

```

import Form from '@/app/ui/invoices/create-form';import Breadcrumbs from '@/app/ui/invoices/breadcrumbs';import { fetchCustomers } from '@/app/lib/data'; export default async function Page() { const customers = await fetchCustomers(); return ( <main> <Breadcrumbs breadcrumbs={[ { label: 'Invoices', href: '/dashboard/invoices' }, { label: 'Create Invoice', href: '/dashboard/invoices/create', active: true, }, ]} /> <Form customers={customers} /> </main> );}

```

Your page is a Server Component that fetches `customers` and passes it to the `<Form>` component. To save time, we've already created the `<Form>` component for you.

Navigate to the `<Form>` component, and you'll see that the form:

- Has one `<select>` (dropdown) element with a list of \*\*customers\*\*.

- Has one `<input>` element for the \*\*amount\*\* with `type="number"`.

- Has two `<input>` elements for the status with `type="radio"`.

- Has one button with `type="submit"`.

On [http://localhost:3000/dashboard/invoices/create](http://localhost:3000/dashboard/invoices/create), you should see the following UI:

![Create invoices page with breadcrumbs and form](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fcreate-invoice-page.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

### [2. Create a Server Action](https://nextjs.org/learn/dashboard-app/mutating-data#2-create-a-server-action)

Great, now let's create a Server Action that is going to be called when the form is submitted.

Navigate to your `lib` directory and create a new file named `actions.ts`. At the top of this file, add the React [`use server`](https://react.dev/reference/react/use-server) directive:

/app/lib/actions.ts

```

'use server';

```

By adding the `'use server'`, you mark all the exported functions within the file as server functions. These server functions can then be imported into Client and Server components, making them extremely versatile.

You can also write Server Actions directly inside Server Components by adding "use server" inside the action. But for this course, we'll keep them all organized in a separate file.

In your `actions.ts` file, create a new async function that accepts `formData`:

/app/lib/actions.ts

```

'use server'; export async function createInvoice(formData: FormData) {}

```

Then, in your `<Form>` component, import the `createInvoice` from your `actions.ts` file. Add a `action` attribute to the `<form>` element, and call the `createInvoice` action.

/app/ui/invoices/create-form.tsx

```

'use client'; import { customerField } from '@/app/lib/definitions';import Link from 'next/link';import { CheckIcon, ClockIcon, CurrencyDollarIcon, UserCircleIcon,} from '@heroicons/react/24/outline';import { Button } from '@/app/ui/button';import { createInvoice } from '@/app/lib/actions'; export default function Form({ customers,}: { customers: customerField[];}) { return ( <form action={createInvoice}> // ... )}

```

> \*\*Good to know\*\*: In HTML, you'd pass a URL to the `action` attribute. This URL would be the destination where your form data should be submitted (usually an API endpoint).

>

> However, in React, the `action` attribute is considered a special prop - meaning React builds on top of it to allow actions to be invoked.

>

> Behind the scenes, Server Actions create a `POST` API endpoint. This is why you don't need to create API endpoints manually when using Server Actions.

### [3. Extract the data from `formData`](https://nextjs.org/learn/dashboard-app/mutating-data#3-extract-the-data-from-formdata)

Back in your `actions.ts` file, you'll need to extract the values of `formData`, there are a [couple of methods](https://developer.mozilla.org/en-US/docs/Web/API/FormData/append) you can use. For this example, let's use the [`.get(name)`](https://developer.mozilla.org/en-US/docs/Web/API/FormData/get) method.

/app/lib/actions.ts

```

'use server'; export async function createInvoice(formData: FormData) { const rawFormData = { customerId: formData.get('customerId'), amount: formData.get('amount'), status: formData.get('status'), }; // Test it out: console.log(rawFormData);}

```

> \*\*Tip:\*\* If you're working with forms that have many fields, you may want to consider using the [`entries()`](https://developer.mozilla.org/en-US/docs/Web/API/FormData/entries) method with JavaScript's [`Object.fromEntries()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Object/entries). For example:

>

> `const rawFormData = Object.fromEntries(formData.entries())`

To check everything is connected correctly, go ahead and try out the form. After submitting, you should see the data you just entered into the form logged in your terminal.

Now that your data is in the shape of an object, it'll be much easier to work with.

### [4. Validate and prepare the data](https://nextjs.org/learn/dashboard-app/mutating-data#4-validate-and-prepare-the-data)

Before sending the form data to your database, you want to ensure it's in the correct format and with the correct types. If you remember from earlier in the course, your invoices table expects data in the following format:

/app/lib/definitions.ts

```

export type Invoice = { id: string; // Will be created on the database customer\_id: string; amount: number; // Stored in cents status: 'pending' | 'paid'; date: string;};

```

So far, you only have the `customer\_id`, `amount`, and `status` from the form.

#### [Type validation and coercion](https://nextjs.org/learn/dashboard-app/mutating-data#type-validation-and-coercion)

It's important to validate that the data from your form aligns with the expected types in your database. For instance, if you add a `console.log` inside your action:

```

console.log(typeof rawFormData.amount);

```

You'll notice that `amount` is of type `string` and not `number`. This is because `input` elements with `type="number"` actually return a string, not a number!

To handle type validation, you have a few options. While you can manually validate types, using a type validation library can save you time and effort. For your example, we'll use [Zod](https://zod.dev/), a TypeScript-first validation library that can simplify this task for you.

In your `actions.ts` file, import Zod and define a schema that matches the shape of your form object. This schema will validate the `formData` before saving it to a database.

/app/lib/actions.ts

```

'use server'; import { z } from 'zod'; const FormSchema = z.object({ id: z.string(), customerId: z.string(), amount: z.coerce.number(), status: z.enum(['pending', 'paid']), date: z.string(),}); const CreateInvoice = FormSchema.omit({ id: true, date: true }); export async function createInvoice(formData: FormData) { // ...}

```

The `amount` field is specifically set to coerce (change) from a string to a number while also validating its type.

You can then pass your `rawFormData` to `CreateInvoice` to validate the types:

/app/lib/actions.ts

```

// ...export async function createInvoice(formData: FormData) { const { customerId, amount, status } = CreateInvoice.parse({ customerId: formData.get('customerId'), amount: formData.get('amount'), status: formData.get('status'), });}

```

#### [Storing values in cents](https://nextjs.org/learn/dashboard-app/mutating-data#storing-values-in-cents)

It's usually good practice to store monetary values in cents in your database to eliminate JavaScript floating-point errors and ensure greater accuracy.

Let's convert the amount into cents:

/app/lib/actions.ts

```

// ...export async function createInvoice(formData: FormData) { const { customerId, amount, status } = CreateInvoice.parse({ customerId: formData.get('customerId'), amount: formData.get('amount'), status: formData.get('status'), }); const amountInCents = amount \* 100;}

```

#### [Creating new dates](https://nextjs.org/learn/dashboard-app/mutating-data#creating-new-dates)

Finally, let's create a new date with the format "YYYY-MM-DD" for the invoice's creation date:

/app/lib/actions.ts

```

// ...export async function createInvoice(formData: FormData) { const { customerId, amount, status } = CreateInvoice.parse({ customerId: formData.get('customerId'), amount: formData.get('amount'), status: formData.get('status'), }); const amountInCents = amount \* 100; const date = new Date().toISOString().split('T')[0];}

```

### [5. Inserting the data into your database](https://nextjs.org/learn/dashboard-app/mutating-data#5-inserting-the-data-into-your-database)

Now that you have all the values you need for your database, you can create an SQL query to insert the new invoice into your database and pass in the variables:

/app/lib/actions.ts

```

import { z } from 'zod';import { sql } from '@vercel/postgres'; // ... export async function createInvoice(formData: FormData) { const { customerId, amount, status } = CreateInvoice.parse({ customerId: formData.get('customerId'), amount: formData.get('amount'), status: formData.get('status'), }); const amountInCents = amount \* 100; const date = new Date().toISOString().split('T')[0]; await sql` INSERT INTO invoices (customer\_id, amount, status, date) VALUES (${customerId}, ${amountInCents}, ${status}, ${date}) `;}

```

Right now, we're not handling any errors. We'll do it in the next chapter. For now, let's move on to the next step.

### [6. Revalidate and redirect](https://nextjs.org/learn/dashboard-app/mutating-data#6-revalidate-and-redirect)

Next.js has a [Client-side Router Cache](https://nextjs.org/docs/app/building-your-application/caching#router-cache) that stores the route segments in the user's browser for a time. Along with [prefetching](https://nextjs.org/docs/app/building-your-application/routing/linking-and-navigating#1-prefetching), this cache ensures that users can quickly navigate between routes while reducing the number of requests made to the server.

Since you're updating the data displayed in the invoices route, you want to clear this cache and trigger a new request to the server. You can do this with the [`revalidatePath`](https://nextjs.org/docs/app/api-reference/functions/revalidatePath) function from Next.js:

/app/lib/actions.ts

```

'use server'; import { z } from 'zod';import { sql } from '@vercel/postgres';import { revalidatePath } from 'next/cache'; // ... export async function createInvoice(formData: FormData) { const { customerId, amount, status } = CreateInvoice.parse({ customerId: formData.get('customerId'), amount: formData.get('amount'), status: formData.get('status'), }); const amountInCents = amount \* 100; const date = new Date().toISOString().split('T')[0]; await sql` INSERT INTO invoices (customer\_id, amount, status, date) VALUES (${customerId}, ${amountInCents}, ${status}, ${date}) `; revalidatePath('/dashboard/invoices');}

```

Once the database has been updated, the `/dashboard/invoices` path will be revalidated, and fresh data will be fetched from the server.

At this point, you also want to redirect the user back to the `/dashboard/invoices` page. You can do this with the [`redirect`](https://nextjs.org/docs/app/api-reference/functions/redirect) function from Next.js:

/app/lib/actions.ts

```

'use server'; import { z } from 'zod';import { sql } from '@vercel/postgres';import { revalidatePath } from 'next/cache';import { redirect } from 'next/navigation'; // ... export async function createInvoice(formData: FormData) { // ... revalidatePath('/dashboard/invoices'); redirect('/dashboard/invoices');}

```

Congratulations! You've just implemented your first Server Action. Test it out by adding a new invoice, if everything is working correctly:

1. You should be redirected to the `/dashboard/invoices` route on submission.

2. You should see the new invoice at the top of the table.

## [Updating an invoice](https://nextjs.org/learn/dashboard-app/mutating-data#updating-an-invoice)

The updating invoice form is similar to the create an invoice form, except you'll need to pass the invoice `id` to update the record in your database. Let's see how you can get and pass the invoice `id`.

These are the steps you'll take to update an invoice:

1. Create a new dynamic route segment with the invoice `id`.

2. Read the invoice `id` from the page params.

3. Fetch the specific invoice from your database.

4. Pre-populate the form with the invoice data.

5. Update the invoice data in your database.

### [1. Create a Dynamic Route Segment with the invoice `id`](https://nextjs.org/learn/dashboard-app/mutating-data#1-create-a-dynamic-route-segment-with-the-invoice-id)

Next.js allows you to create [Dynamic Route Segments](https://nextjs.org/docs/app/building-your-application/routing/dynamic-routes) when you don't know the exact segment name and want to create routes based on data. This could be blog post titles, product pages, etc. You can create dynamic route segments by wrapping a folder's name in square brackets. For example, `[id]`, `[post]` or `[slug]`.

In your `/invoices` folder, create a new dynamic route called `[id]`, then a new route called `edit` with a `page.tsx` file. Your file structure should look like this:

![Invoices folder with a nested [id] folder, and an edit folder inside it](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fedit-invoice-route.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

In your `<Table>` component, notice there's a `<UpdateInvoice />` button that receives the invoice's `id` from the table records.

/app/ui/invoices/table.tsx

```

export default async function InvoicesTable({ query, currentPage,}: { query: string; currentPage: number;}) { return ( // ... <td className="flex justify-end gap-2 whitespace-nowrap px-6 py-4 text-sm"> <UpdateInvoice id={invoice.id} /> <DeleteInvoice id={invoice.id} /> </td> // ... );}

```

Navigate to your `<UpdateInvoice />` component, and update the `href` of the `Link` to accept the `id` prop. You can use template literals to link to a dynamic route segment:

/app/ui/invoices/buttons.tsx

```

import { PencilIcon, PlusIcon, TrashIcon } from '@heroicons/react/24/outline';import Link from 'next/link'; // ... export function UpdateInvoice({ id }: { id: string }) { return ( <Link href={`/dashboard/invoices/${id}/edit`} className="rounded-md border p-2 hover:bg-gray-100" > <PencilIcon className="w-5" /> </Link> );}

```

### [2. Read the invoice `id` from page `params`](https://nextjs.org/learn/dashboard-app/mutating-data#2-read-the-invoice-id-from-page-params)

Back on your `<Page>` component, paste the following code:

/app/dashboard/invoices/[id]/edit/page.tsx

```

import Form from '@/app/ui/invoices/edit-form';import Breadcrumbs from '@/app/ui/invoices/breadcrumbs';import { fetchCustomers } from '@/app/lib/data'; export default async function Page() { return ( <main> <Breadcrumbs breadcrumbs={[ { label: 'Invoices', href: '/dashboard/invoices' }, { label: 'Edit Invoice', href: `/dashboard/invoices/${id}/edit`, active: true, }, ]} /> <Form invoice={invoice} customers={customers} /> </main> );}

```

Notice how it's similar to your `/create` invoice page, except it imports a different form (from the `edit-form.tsx` file). This form should be \*\*pre-populated\*\* with a `defaultValue` for the customer's name, invoice amount, and status. To pre-populate the form fields, you need to fetch the specific invoice using `id`.

In addition to `searchParams`, page components also accept a prop called `params` which you can use to access the `id`. Update your `<Page>` component to receive the prop:

/app/dashboard/invoices/[id]/edit/page.tsx

```

import Form from '@/app/ui/invoices/edit-form';import Breadcrumbs from '@/app/ui/invoices/breadcrumbs';import { fetchCustomers } from '@/app/lib/data'; export default async function Page({ params }: { params: { id: string } }) { const id = params.id; // ...}

```

### [3. Fetch the specific invoice](https://nextjs.org/learn/dashboard-app/mutating-data#3-fetch-the-specific-invoice)

Then:

- Import a new function called `fetchInvoiceById` and pass the `id` as an argument.

- Import `fetchCustomers` to fetch the customer names for the dropdown.

You can use `Promise.all` to fetch both the invoice and customers in parallel:

/dashboard/invoices/[id]/edit/page.tsx

```

import Form from '@/app/ui/invoices/edit-form';import Breadcrumbs from '@/app/ui/invoices/breadcrumbs';import { fetchInvoiceById, fetchCustomers } from '@/app/lib/data'; export default async function Page({ params }: { params: { id: string } }) { const id = params.id; const [invoice, customers] = await Promise.all([ fetchInvoiceById(id), fetchCustomers(), ]); // ...}

```

You will see a temporary TS error for the `invoices` prop in your terminal because invoices could be potentially undefined. Don't worry about it for now, you'll resolve it in the next chapter when you add error handling.

Great! Now, test that everything is wired correctly. Visit [http://localhost:3000/dashboard/invoices](http://localhost:3000/dashboard/invoices) and click on the Pencil icon to edit an invoice. After navigation, you should see a form that is pre-populated with the invoice details:

![Edit invoices page with breadcrumbs and form](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fedit-invoice-page.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

The URL should also be updated with an `id` as follows: `http://localhost:3000/dashboard/invoice/uuid/edit`

> \*\*UUIDs vs. Auto-incrementing Keys\*\*

>

> We use UUIDs instead of incrementing keys (e.g., 1, 2, 3, etc.). This makes the URL longer; however, UUIDs eliminate the risk of ID collision, are globally unique, and reduce the risk of enumeration attacks - making them ideal for large databases.

>

> However, if you prefer cleaner URLs, you might prefer to use auto-incrementing keys.

### [4. Pass the `id` to the Server Action](https://nextjs.org/learn/dashboard-app/mutating-data#4-pass-the-id-to-the-server-action)

Lastly, you want to pass the `id` to the Server Action so you can update the right record in your database. You \*\*cannot\*\* pass the `id` as an argument like so:

/app/ui/invoices/edit-form.tsx

```

// Passing an id as argument won't work<form action={updateInvoice(id)}>

```

Instead, you can pass `id` to the Server Action using JS `bind`. This will ensure that any values passed to the Server Action are encoded.

/app/ui/invoices/edit-form.tsx

```

// ...import { updateInvoice } from '@/app/lib/actions'; export default function EditInvoiceForm({ invoice, customers,}: { invoice: InvoiceForm; customers: CustomerField[];}) { const updateInvoiceWithId = updateInvoice.bind(null, invoice.id); return ( <form action={updateInvoiceWithId}> <input type="hidden" name="id" value={invoice.id} /> </form> );}

```

> \*\*Note:\*\* Using a hidden input field in your form also works (e.g. `<input type="hidden" name="id" value={invoice.id} />`). However, the values will appear as full text in the HTML source, which is not ideal for sensitive data like IDs.

Then, in your `actions.ts` file, create a new action, `updateInvoice`:

/app/lib/actions.ts

```

// Use Zod to update the expected typesconst UpdateInvoice = FormSchema.omit({ id: true, date: true }); // ... export async function updateInvoice(id: string, formData: FormData) { const { customerId, amount, status } = UpdateInvoice.parse({ customerId: formData.get('customerId'), amount: formData.get('amount'), status: formData.get('status'), }); const amountInCents = amount \* 100; await sql` UPDATE invoices SET customer\_id = ${customerId}, amount = ${amountInCents}, status = ${status} WHERE id = ${id} `; revalidatePath('/dashboard/invoices'); redirect('/dashboard/invoices');}

```

Similarly to the `createInvoice` action, here you are:

1. Extracting the data from `formData`.

2. Validating the types with Zod.

3. Converting the amount to cents.

4. Passing the variables to your SQL query.

5. Calling `revalidatePath` to clear the client cache and make a new server request.

6. Calling `redirect` to redirect the user to the invoice's page.

Test it out by editing an invoice. After submitting the form, you should be redirected to the invoices page, and the invoice should be updated.

## [Deleting an invoice](https://nextjs.org/learn/dashboard-app/mutating-data#deleting-an-invoice)

To delete an invoice using a Server Action, wrap the delete button in a `<form>` element and pass the `id` to the Server Action using `bind`:

/app/ui/invoices/buttons.tsx

```

import { deleteInvoice } from '@/app/lib/actions'; // ... export function DeleteInvoice({ id }: { id: string }) { const deleteInvoiceWithId = deleteInvoice.bind(null, id); return ( <form action={deleteInvoiceWithId}> <button className="rounded-md border p-2 hover:bg-gray-100"> <span className="sr-only">Delete</span> <TrashIcon className="w-4" /> </button> </form> );}

```

Inside your `actions.ts` file, create a new action called `deleteInvoice`.

/app/lib/actions.ts

```

export async function deleteInvoice(id: string) { await sql`DELETE FROM invoices WHERE id = ${id}`; revalidatePath('/dashboard/invoices');}

```

Since this action is being called in the `/dashboard/invoices` path, you don't need to call `redirect`. Calling `revalidatePath` will trigger a new server request and re-render the table.

## [Further reading](https://nextjs.org/learn/dashboard-app/mutating-data#further-reading)

In this chapter, you learned how to use Server Actions to mutate data. You also learned how to use the `revalidatePath` API to revalidate the Next.js cache and `redirect` to redirect the user to a new page.

You can also read more about [security with Server Actions](https://nextjs.org/blog/security-nextjs-server-components-actions) for additional learning.

# Handling Errors

In the previous chapter, you learned how to mutate data using Server Actions. Let's see how you can handle errors \_gracefully\_ using JavaScript's `try/catch` statements and Next.js APIs.

In this chapter...

Here are the topics we’ll cover

How to use the special `error.tsx` file to catch errors in your route segments, and show a fallback UI to the user.

How to use the `notFound` function and `not-found` file to handle 404 errors (for resources that don’t exist).

## [Adding `try/catch` to Server Actions](https://nextjs.org/learn/dashboard-app/error-handling#adding-trycatch-to-server-actions)

First, let's add JavaScript's `try/catch` statements to your Server Actions to allow you to handle errors gracefully.

If you know how to do this, spend a few minutes updating your Server Actions, or you can copy the code below:

Reveal the solution

Reveal the solution

Reveal the solution

Note how `redirect` is being called outside of the `try/catch` block. This is because `redirect` works by throwing an error, which would be caught by the `catch` block. To avoid this, you can call `redirect` \*\*after\*\* `try/catch`. `redirect` would only be reachable if `try` is successful.

Now, let's check what happens when an error is thrown in your Server Action. You can do this by throwing an error earlier. For example, in the `deleteInvoice` action, throw an error at the top of the function:

/app/lib/actions.ts

```

export async function deleteInvoice(id: string) { throw new Error('Failed to Delete Invoice'); // Unreachable code block try { await sql`DELETE FROM invoices WHERE id = ${id}`; revalidatePath('/dashboard/invoices'); return { message: 'Deleted Invoice' }; } catch (error) { return { message: 'Database Error: Failed to Delete Invoice' }; }}

```

When you try to delete an invoice, you should see an error on localhost.

Seeing these errors are helpful while developing as you can catch any potential problems early. However, you also want to show errors to the user to avoid an abrupt failure and allow your application to continue running.

This is where Next.js [`error.tsx`](https://nextjs.org/docs/app/api-reference/file-conventions/error) file comes in.

## [Handling all errors with `error.tsx`](https://nextjs.org/learn/dashboard-app/error-handling#handling-all-errors-with-errortsx)

The `error.tsx` file can be used to define a UI boundary for a route segment. It serves as a \*\*catch-all\*\* for unexpected errors and allows you to display a fallback UI to your users.

Inside your `/dashboard/invoices` folder, create a new file called `error.tsx` and paste the following code:

/dashboard/invoices/error.tsx

```

'use client'; import { useEffect } from 'react'; export default function Error({ error, reset,}: { error: Error & { digest?: string }; reset: () => void;}) { useEffect(() => { // Optionally log the error to an error reporting service console.error(error); }, [error]); return ( <main className="flex h-full flex-col items-center justify-center"> <h2 className="text-center">Something went wrong!</h2> <button className="mt-4 rounded-md bg-blue-500 px-4 py-2 text-sm text-white transition-colors hover:bg-blue-400" onClick={ // Attempt to recover by trying to re-render the invoices route () => reset() } > Try again </button> </main> );}

```

There are a few things you'll notice about the code above:

- \*\*"use client"\*\* - `error.tsx` needs to be a Client Component.

- It accepts two props:

- `error`: This object is an instance of JavaScript's native [`Error`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Error) object.

- `reset`: This is a function to reset the error boundary. When executed, the function will try to re-render the route segment.

When you try to delete an invoice again, you should see the following UI:

![The error.tsx file showing the props it accepts](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Ferror-page.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

## [Handling 404 errors with the `notFound` function](https://nextjs.org/learn/dashboard-app/error-handling#handling-404-errors-with-the-notfound-function)

Another way you can handle errors gracefully is by using the `notFound` function. While `error.tsx` is useful for catching \*\*all\*\* errors, `notFound` can be used when you try to fetch a resource that doesn't exist.

For example, visit [http://localhost:3000/dashboard/invoices/2e94d1ed-d220-449f-9f11-f0bbceed9645/edit](http://localhost:3000/dashboard/invoices/2e94d1ed-d220-449f-9f11-f0bbceed9645/edit).

This is a fake UUID that doesn't exist in your database.

You'll immediately see `error.tsx` kicks in because this is a child route of `/invoices` where `error.tsx` is defined.

However, if you want to be more specific, you can show a 404 error to tell the user the resource they're trying to access hasn't been found.

You can confirm that the resource hasn't been found by going into your `fetchInvoiceById` function in `data.ts`, and console logging the returned `invoice`:

/app/lib/data.ts

```

export async function fetchInvoiceById(id: string) { noStore(); try { // ... console.log(invoice); // Invoice is an empty array [] return invoice[0]; } catch (error) { console.error('Database Error:', error); throw new Error('Failed to fetch invoice.'); }}

```

Now that you know the invoice doesn't exist in your database, let's use `notFound` to handle it. Navigate to `/dashboard/invoices/[id]/edit/page.tsx`, and import `{ notFound }` from `'next/navigation'`.

Then, you can use a conditional to invoke `notFound` if the invoice doesn't exist:

/dashboard/invoices/[id]/edit/page.tsx

```

import { fetchInvoiceById, fetchCustomers } from '@/app/lib/data';import { updateInvoice } from '@/app/lib/actions';import { notFound } from 'next/navigation'; export default async function Page({ params }: { params: { id: string } }) { const id = params.id; const [invoice, customers] = await Promise.all([ fetchInvoiceById(id), fetchCustomers(), ]); if (!invoice) { notFound(); } // ...}

```

Perfect! `<Page>` will now throw an error if a specific invoice is not found. To show an error UI to the user. Create a `not-found.tsx` file inside the `/edit` folder.

![The not-found.tsx file inside the edit folder](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fnot-found-file.png&w=3840&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

Then, inside the `not-found.tsx` file, paste the following the code:

/dashboard/invoices/[id]/edit/not-found.tsx

```

import Link from 'next/link';import { FaceFrownIcon } from '@heroicons/react/24/outline'; export default function NotFound() { return ( <main className="flex h-full flex-col items-center justify-center gap-2"> <FaceFrownIcon className="w-10 text-gray-400" /> <h2 className="text-xl font-semibold">404 Not Found</h2> <p>Could not find the requested invoice.</p> <Link href="/dashboard/invoices" className="mt-4 rounded-md bg-blue-500 px-4 py-2 text-sm text-white transition-colors hover:bg-blue-400" > Go Back </Link> </main> );}

```

Refresh the route, and you should now see the following UI:

![404 Not Found Page](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2F404-not-found-page.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

That's something to keep in mind, `notFound` will take precedence over `error.tsx`, so you can reach out for it when you want to handle more specific errors!

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

Which file in Next.js serves as a catch-all for unexpected errors in your route segments?

## [Further reading](https://nextjs.org/learn/dashboard-app/error-handling#further-reading)

To learn more about error handling in Next.js, check out the following documentation:

- [Error Handling](https://nextjs.org/docs/app/building-your-application/routing/error-handling)

- [`error.js` API Reference](https://nextjs.org/docs/app/api-reference/file-conventions/error)

- [`notFound()` API Reference](https://nextjs.org/docs/app/api-reference/functions/not-found)

- [`not-found.js` API Reference](https://nextjs.org/docs/app/api-reference/file-conventions/not-found)

## You've Completed Chapter 13

Nice, you're now able to handle errors gracefully in your application.

# Improving Accessibility

In the previous chapter, we looked at how to catch errors (including 404 errors) and display a fallback to the user. However, we still need to discuss another piece of the puzzle: form validation. Let's see how to implement server-side validation with Server Actions, and how you can show form errors using the `useFormState` hook - while keeping accessibility in mind!

In this chapter...

Here are the topics we’ll cover

How to use `eslint-plugin-jsx-a11y` with Next.js to implement accessibility best practices.

How to implement server-side form validation.

How to use the React `useFormState` hook to handle form errors, and display them to the user.

## [What is accessibility?](https://nextjs.org/learn/dashboard-app/improving-accessibility#what-is-accessibility)

Accessibility refers to designing and implementing web applications that everyone can use, including those with disabilities. It's a vast topic that covers many areas, such as keyboard navigation, semantic HTML, images, colors, videos, etc.

While we won't go in-depth into accessibility in this course, we'll discuss the accessibility features available in Next.js and some common practices to make your applications more accessible.

> If you'd like to learn more about accessibility, we recommend the [Learn Accessibility](https://web.dev/learn/accessibility/) course by [web.dev](https://web.dev/).

## [Using the ESLint accessibility plugin in Next.js](https://nextjs.org/learn/dashboard-app/improving-accessibility#using-the-eslint-accessibility-plugin-in-nextjs)

By default, Next.js includes the [`eslint-plugin-jsx-a11y`](https://www.npmjs.com/package/eslint-plugin-jsx-a11y) plugin to help catch accessibility issues early. For example, this plugin warns if you have images without `alt` text, use the `aria-\*` and `role` attributes incorrectly, and more.

Let's see how this works!

Add `next lint` as a script in your `package.json` file:

/package.json

```

"scripts": { "build": "next build", "dev": "next dev", "seed": "node -r dotenv/config ./scripts/seed.js", "start": "next start", "lint": "next lint"},

```

Then run `npm run lint` in your terminal:

Terminal

```

npm run lint

```

You should see the following warning:

Terminal

```

✔ No ESLint warnings or errors

```

However, what would happen if you had an image without an `alt` tag? Let's find out!

Go to `/app/ui/invoices/table.tsx` and remove the `alt` tag from the image. You can use your editor's search feature to quickly find the image tag:

/app/ui/invoices/table.tsx

```

<Image src={invoice.image\_url} className="rounded-full" width={28} height={28} alt={`${invoice.name}'s profile picture`} // Delete this line/>

```

Now run `npm run lint` again, and you should see the following warning:

Terminal

```

./app/ui/invoices/table.tsx45:25 Warning: Image elements must have an alt prop,either with meaningful text, or an empty string for decorative images. jsx-a11y/alt-text

```

If you tried to deploy your application to Vercel, the warning would also show up in the build logs. This is because `next lint` runs as part of the build process. So you can run `lint` locally to catch accessibility issues before deploying your application.

## [Improving form accessibility](https://nextjs.org/learn/dashboard-app/improving-accessibility#improving-form-accessibility)

There are three things we're already doing to improve accessibility in our forms:

- \*\*Semantic HTML\*\*: Using semantic elements (`<input>`, `<option>`, etc) instead of `<div>`. This allows assistive technologies (AT) to focus on the input elements and provide appropriate contextual information to the user, making the form easier to navigate and understand.

- \*\*Labelling\*\*: Including `<label>` and the `htmlFor` attribute ensures that each form field has a descriptive text label. This improves AT support by providing context and also enhances usability by allowing users to click on the label to focus on the corresponding input field.

- \*\*Focus Outline\*\*: The fields are properly styled to show an outline when they are in focus. This is critical for accessibility as it visually indicates the active element on the page, helping both keyboard and screen reader users to understand where they are on the form. You can verify this by pressing `tab`.

These practices lay a good foundation for making your forms more accessible to many users. However, they don't address \*\*form validation\*\* and \*\*errors\*\*.

## [Form validation](https://nextjs.org/learn/dashboard-app/improving-accessibility#form-validation)

Go to [http://localhost:3000/dashboard/invoices/create](http://localhost:3000/dashboard/invoices/create), and submit an empty form. What happens?

You get an error! This is because you're sending empty form values to your Server Action. You can prevent this by validating your form on the client or the server.

### [Client-Side validation](https://nextjs.org/learn/dashboard-app/improving-accessibility#client-side-validation)

There are a couple of ways you can validate forms on the client. The simplest would be to rely on the form validation provided by the browser by adding the `required` attribute to the `<input>` and `<select>` elements in your forms. For example:

/app/ui/invoices/create-form.tsx

```

<input id="amount" name="amount" type="number" placeholder="Enter USD amount" className="peer block w-full rounded-md border border-gray-200 py-2 pl-10 text-sm outline-2 placeholder:text-gray-500" required/>

```

Submit the form again, and you should now see the browser a warning if you try to submit a form with empty values.

This approach is generally okay because some ATs support browser validation.

An alternative to client-side validation is server-side validation. Let's see how you can implement it in the next section. For now, delete the `required` attributes if you added them.

### [Server-Side validation](https://nextjs.org/learn/dashboard-app/improving-accessibility#server-side-validation)

By validating forms on the server, you can:

- Ensure your data is in the expected format before sending it to your database.

- Reduce the risk of malicious users bypassing client-side validation.

- Have one source of truth for what is considered \_valid\_ data.

In your `create-form.tsx` component, import the `useFormState` hook from `react-dom`. Since `useFormState` is a hook, you will need to turn your form into a Client Component using `"use client"` directive:

/app/ui/invoices/create-form.tsx

```

'use client'; // ...import { useFormState } from 'react-dom';

```

Inside your Form Component, the `useFormState` hook:

- Takes two arguments: `(action, initialState)`.

- Returns two values: `[state, dispatch]` - the form state, and a dispatch function (similar to [useReducer](https://react.dev/reference/react/useReducer))

Pass your `createInvoice` action as an argument of `useFormState`, and inside your `<form action={}>` attribute, call `dispatch`.

/app/ui/invoices/create-form.tsx

```

// ...import { useFormState } from 'react-dom'; export default function Form({ customers }: { customers: CustomerField[] }) { const [state, dispatch] = useFormState(createInvoice, initialState); return <form action={dispatch}>...</form>;}

```

The `initialState` can be anything you define, in this case, create an object with two empty keys: `message` and `errors`.

/app/ui/invoices/create-form.tsx

```

// ...import { useFormState } from 'react-dom'; export default function Form({ customers }: { customers: CustomerField[] }) { const initialState = { message: null, errors: {} }; const [state, dispatch] = useFormState(createInvoice, initialState); return <form action={dispatch}>...</form>;}

```

This may seem confusing initially, but it'll make more sense once you update the server action. Let's do that now.

In your `action.ts` file, you can use Zod to validate form data. Update your `FormSchema` as follows:

/app/lib/action.ts

```

const FormSchema = z.object({ id: z.string(), customerId: z.string({ invalid\_type\_error: 'Please select a customer.', }), amount: z.coerce .number() .gt(0, { message: 'Please enter an amount greater than $0.' }), status: z.enum(['pending', 'paid'], { invalid\_type\_error: 'Please select an invoice status.', }), date: z.string(),});

```

- `customerId` - Zod already throws an error if the customer field is empty as it expects a type `string`. But let's add a friendly message if the user doesn't select a customer.

- `amount` - Since you are coercing the amount type from `string` to `number`, it'll default to zero if the string is empty. Let's tell Zod we always want the amount greater than 0 with the `.gt()` function.

- `status` - Zod already throws an error if the status field is empty as it expects "pending" or "paid". Let's also add a friendly message if the user doesn't select a status.

Next, update your `createInvoice` action to accept two parameters:

/app/lib/actions.ts

```

// This is temporary until @types/react-dom is updatedexport type State = { errors?: { customerId?: string[]; amount?: string[]; status?: string[]; }; message?: string | null;}; export async function createInvoice(prevState: State, formData: FormData) { // ...}

```

- `formData` - same as before.

- `prevState` - contains the state passed from the `useFormState` hook. You won't be using it in the action in this example, but it's a required prop.

Then, change the Zod `parse()` function to `safeParse()`:

/app/lib/actions.ts

```

export async function createInvoice(prevState: State, formData: FormData) { // Validate form fields using Zod const validatedFields = CreateInvoice.safeParse({ customerId: formData.get('customerId'), amount: formData.get('amount'), status: formData.get('status'), }); // ...}

```

`safeParse()` will return an object containing either a `success` or `error` field. This will help handle validation more gracefully without having put this logic inside the `try/catch` block.

Before sending the information to your database, check if the form fields were validated correctly with a conditional:

/app/lib/actions.ts

```

export async function createInvoice(prevState: State, formData: FormData) { // Validate form fields using Zod const validatedFields = CreateInvoice.safeParse({ customerId: formData.get('customerId'), amount: formData.get('amount'), status: formData.get('status'), }); // If form validation fails, return errors early. Otherwise, continue. if (!validatedFields.success) { return { errors: validatedFields.error.flatten().fieldErrors, message: 'Missing Fields. Failed to Create Invoice.', }; } // ...}

```

If `validatedFields` isn't successful, we return the function early with the error messages from Zod.

> \*\*Tip:\*\* console.log `validatedFields` and submit an empty form to see the shape of it.

Finally, since you're handling form validation separately, outside your try/catch block, you can return a specific message for any database errors, your final code should look like this:

/app/lib/actions.ts

```

export async function createInvoice(prevState: State, formData: FormData) { // Validate form using Zod const validatedFields = CreateInvoice.safeParse({ customerId: formData.get('customerId'), amount: formData.get('amount'), status: formData.get('status'), }); // If form validation fails, return errors early. Otherwise, continue. if (!validatedFields.success) { return { errors: validatedFields.error.flatten().fieldErrors, message: 'Missing Fields. Failed to Create Invoice.', }; } // Prepare data for insertion into the database const { customerId, amount, status } = validatedFields.data; const amountInCents = amount \* 100; const date = new Date().toISOString().split('T')[0]; // Insert data into the database try { await sql` INSERT INTO invoices (customer\_id, amount, status, date) VALUES (${customerId}, ${amountInCents}, ${status}, ${date}) `; } catch (error) { // If a database error occurs, return a more specific error. return { message: 'Database Error: Failed to Create Invoice.', }; } // Revalidate the cache for the invoices page and redirect the user. revalidatePath('/dashboard/invoices'); redirect('/dashboard/invoices');}

```

Great, now let's display the errors in your form component. Back in the `create-form.tsx` component, you can access the errors using the form `state`.

Add a \*\*ternary operator\*\* that checks for each specific error. For example, after the customer's field, you can add:

/app/ui/invoices/create-form.tsx

```

<form action={dispatch}> <div className="rounded-md bg-gray-50 p-4 md:p-6"> {/\* Customer Name \*/} <div className="mb-4"> <label htmlFor="customer" className="mb-2 block text-sm font-medium"> Choose customer </label> <div className="relative"> <select id="customer" name="customerId" className="peer block w-full rounded-md border border-gray-200 py-2 pl-10 text-sm outline-2 placeholder:text-gray-500" defaultValue="" aria-describedby="customer-error" > <option value="" disabled> Select a customer </option> {customerNames.map((name) => ( <option key={name.id} value={name.id}> {name.name} </option> ))} </select> <UserCircleIcon className="pointer-events-none absolute left-3 top-1/2 h-[18px] w-[18px] -translate-y-1/2 text-gray-500" /> </div> <div id="customer-error" aria-live="polite" aria-atomic="true"> {state.errors?.customerId && state.errors.customerId.map((error: string) => ( <p className="mt-2 text-sm text-red-500" key={error}> {error} </p> ))} </div> </div> // ... </div></form>

```

> \*\*Tip:\*\* You can console.log `state` inside your component and check if everything is wired correctly. Check the console in Dev Tools as your form is now a Client Component.

In the code above, you're also adding the following aria labels:

- `aria-describedby="customer-error"`: This establishes a relationship between the `select` element and the error message container. It indicates that the container with `id="customer-error"` describes the `select` element. Screen readers will read this description when the user interacts with the `select` box to notify them of errors.

- `id="customer-error"`: This `id` attribute uniquely identifies the HTML element that holds the error message for the `select` input. This is necessary for `aria-describedby` to establish the relationship.

- `aria-live="polite"`: The screen reader should politely notify the user when the error inside the `div` is updated. When the content changes (e.g. when a user corrects an error), the screen reader will announce these changes, but only when the user is idle so as not to interrupt them.

# [Practice: Adding aria labels](https://nextjs.org/learn/dashboard-app/improving-accessibility#practice-adding-aria-labels)

Using the example above, add errors to your remaining form fields. You should also show a message at the bottom of the form if any fields are missing. Your UI should look like this:

![Create invoice form showing error messages for each field.](https://nextjs.org/\_next/image?url=%2Flearn%2Fdark%2Fform-validation-page.png&w=1920&q=75&dpl=dpl\_A8AJ5FwmLMU94D4JXDc9U5wQRc6m)

Once you're ready, run `npm run lint` to check if you're using the aria labels correctly.

If you'd like to challenge yourself, take the knowledge you've learned in this chapter and add form validation to the `edit-form.tsx` component.

You'll need to:

- Add `useFormState` to your `edit-form.tsx` component.

- Edit the `updateInvoice` action to handle validation errors from Zod.

- Display the errors in your component, and add aria labels to improve accessibility.

Once you're ready, expand the code snippet below to see the solution:

15

Chapter 15

# Adding Authentication

In the previous chapter, you finished building the invoices routes by adding form validation and improving accessibility. In this chapter, you'll be adding authentication to your dashboard.

In this chapter...

Here are the topics we’ll cover

What is authentication.

How to add authentication to your app using NextAuth.js.

How to use Middleware to redirect users and protect your routes.

How to use React's `useFormStatus` and `useFormState` to handle pending states and form errors.

## [What is authentication?](https://nextjs.org/learn/dashboard-app/adding-authentication#what-is-authentication)

Authentication is a key part of many web applications today. It's how a system checks if the user is who they say they are.

A secure website often uses multiple ways to check a user's identity. For instance, after entering your username and password, the site may send a verification code to your device or use an external app like Google Authenticator. This 2-factor authentication (2FA) helps increase security. Even if someone learns your password, they can't access your account without your unique token.

### [Authentication vs. Authorization](https://nextjs.org/learn/dashboard-app/adding-authentication#authentication-vs-authorization)

In web development, authentication and authorization serve different roles:

- \*\*Authentication\*\* is about making sure the user is who they say they are. You're proving your identity with something you have like a username and password.

- \*\*Authorization\*\* is the next step. Once a user's identity is confirmed, authorization decides what parts of the application they are allowed to use.

So, authentication checks who you are, and authorization determines what you can do or access in the application.

### It’s time to take a quiz!

Test your knowledge and see what you’ve just learned.

Which of the following best describes the difference between authentication and authorization?

## [Creating the login route](https://nextjs.org/learn/dashboard-app/adding-authentication#creating-the-login-route)

Start by creating a new route in your application called `/login` and paste the following code:

/app/login/page.tsx

```

import AcmeLogo from '@/app/ui/acme-logo';import LoginForm from '@/app/ui/login-form'; export default function LoginPage() { return ( <main className="flex items-center justify-center md:h-screen"> <div className="relative mx-auto flex w-full max-w-[400px] flex-col space-y-2.5 p-4 md:-mt-32"> <div className="flex h-20 w-full items-end rounded-lg bg-blue-500 p-3 md:h-36"> <div className="w-32 text-white md:w-36"> <AcmeLogo /> </div> </div> <LoginForm /> </div> </main> );}

```

You'll notice the page imports `<LoginForm />`, which you'll update later in the chapter.

## [NextAuth.js](https://nextjs.org/learn/dashboard-app/adding-authentication#nextauthjs)

We will be using [NextAuth.js](https://nextjs.authjs.dev/) to add authentication to your application. NextAuth.js abstracts away much of the complexity involved in managing sessions, sign-in and sign-out, and other aspects of authentication. While you can manually implement these features, the process can be time-consuming and error-prone. NextAuth.js simplifies the process, providing a unified solution for auth in Next.js applications.

## [Setting up NextAuth.js](https://nextjs.org/learn/dashboard-app/adding-authentication#setting-up-nextauthjs)

Install NextAuth.js by running the following command in your terminal:

Terminal

```

npm install next-auth@beta

```

Here, you're installing the `beta` version of NextAuth.js, which is compatible with Next.js 14.

Next, generate a secret key for your application. This key is used to encrypt cookies, ensuring the security of user sessions. You can do this by running the following command in your terminal:

Terminal

```

openssl rand -base64 32

```

Then, in your `.env` file, add your generated key to the `AUTH\_SECRET` variable:

.env

```

AUTH\_SECRET=your-secret-key

```

For auth to work in production, you'll need to update your environment variables in your Vercel project too. Check out this [guide](https://vercel.com/docs/projects/environment-variables) on how to add environment variables on Vercel.

### [Adding the pages option](https://nextjs.org/learn/dashboard-app/adding-authentication#adding-the-pages-option)

Create an `auth.config.ts` file at the root of our project that exports an `authConfig` object. This object will contain the configuration options for NextAuth.js. For now, it will only contain the `pages` option:

/auth.config.ts

```

import type { NextAuthConfig } from 'next-auth'; export const authConfig = { pages: { signIn: '/login', },};

```

You can use the `pages` option to specify the route for custom sign-in, sign-out, and error pages. This is not required, but by adding `signIn: '/login'` into our `pages` option, the user will be redirected to our custom login page, rather than the NextAuth.js default page.

## [Protecting your routes with Next.js Middleware](https://nextjs.org/learn/dashboard-app/adding-authentication#protecting-your-routes-with-nextjs-middleware)

Next, add the logic to protect your routes. This will prevent users from accessing the dashboard pages unless they are logged in.

/auth.config.ts

```

import type { NextAuthConfig } from 'next-auth'; export const authConfig = { pages: { signIn: '/login', }, callbacks: { authorized({ auth, request: { nextUrl } }) { const isLoggedIn = !!auth?.user; const isOnDashboard = nextUrl.pathname.startsWith('/dashboard'); if (isOnDashboard) { if (isLoggedIn) return true; return false; // Redirect unauthenticated users to login page } else if (isLoggedIn) { return Response.redirect(new URL('/dashboard', nextUrl)); } return true; }, }, providers: [], // Add providers with an empty array for now} satisfies NextAuthConfig;

```

The `authorized` callback is used to verify if the request is authorized to access a page via [Next.js Middleware](https://nextjs.org/docs/app/building-your-application/routing/middleware). It is called before a request is completed, and it receives an object with the `auth` and `request` properties. The `auth` property contains the user's session, and the `request` property contains the incoming request.

The `providers` option is an array where you list different login options. For now, it's an empty array to satisfy NextAuth config. You'll learn more about it in the [Adding the Credentials provider](https://nextjs.org/learn/dashboard-app/adding-authentication#adding-the-credentials-provider) section.

Next, you will need to import the `authConfig` object into a Middleware file. In the root of your project, create a file called `middleware.ts` and paste the following code:

/middleware.ts

```

import NextAuth from 'next-auth';import { authConfig } from './auth.config'; export default NextAuth(authConfig).auth; export const config = { // https://nextjs.org/docs/app/building-your-application/routing/middleware#matcher matcher: ['/((?!api|\_next/static|\_next/image|.\*\\.png$).\*)'],};

```

Here you're initializing NextAuth.js with the `authConfig` object and exporting the `auth` property. You're also using the `matcher` option from Middleware to specify that it should run on specific paths.

The advantage of employing Middleware for this task is that the protected routes will not even start rendering until the Middleware verifies the authentication, enhancing both the security and performance of your application.

### [Password hashing](https://nextjs.org/learn/dashboard-app/adding-authentication#password-hashing)

It's good practice to \*\*hash\*\* passwords before storing them in a database. Hashing converts a password into a fixed-length string of characters, which appears random, providing a layer of security even if the user's data is exposed.

In your `seed.js` file, you used a package called `bcrypt` to hash the user's password before storing it in the database. You will use it \_again\_ later in this chapter to compare that the password entered by the user matches the one in the database. However, you will need to create a separate file for the `bcrypt` package. This is because `bcrypt` relies on Node.js APIs not available in Next.js Middleware.

Create a new file called `auth.ts` that spreads your `authConfig` object:

/auth.ts

```

import NextAuth from 'next-auth';import { authConfig } from './auth.config'; export const { auth, signIn, signOut } = NextAuth({ ...authConfig,});

```

### [Adding the Credentials provider](https://nextjs.org/learn/dashboard-app/adding-authentication#adding-the-credentials-provider)

Next, you will need to add the `providers` option for NextAuth.js. `providers` is an array where you list different login options such as Google or GitHub. For this course, we will focus on using the [Credentials provider](https://authjs.dev/getting-started/providers/credentials-tutorial) only.

The Credentials provider allows users to log in with a username and a password.

/auth.ts

```

import NextAuth from 'next-auth';import { authConfig } from './auth.config';import Credentials from 'next-auth/providers/credentials'; export const { auth, signIn, signOut } = NextAuth({ ...authConfig, providers: [Credentials({})],});

```

> \*\*Good to know:\*\*

>

> Although we're using the Credentials provider, it's generally recommended to use alternative providers such as [OAuth](https://authjs.dev/getting-started/providers/oauth-tutorial) or [email](https://authjs.dev/getting-started/providers/email-tutorial) providers. See the [NextAuth.js docs](https://authjs.dev/getting-started/providers) for a full list of options.

### [Adding the sign in functionality](https://nextjs.org/learn/dashboard-app/adding-authentication#adding-the-sign-in-functionality)

You can use the `authorize` function to handle the authentication logic. Similarly to Server Actions, you can use `zod` to validate the email and password before checking if the user exists in the database:

/auth.ts

```

import NextAuth from 'next-auth';import { authConfig } from './auth.config';import Credentials from 'next-auth/providers/credentials';import { z } from 'zod'; export const { auth, signIn, signOut } = NextAuth({ ...authConfig, providers: [ Credentials({ async authorize(credentials) { const parsedCredentials = z .object({ email: z.string().email(), password: z.string().min(6) }) .safeParse(credentials); }, }), ],});

```

After validating the credentials, create a new `getUser` function that queries the user from the database.

/auth.ts

```

import NextAuth from 'next-auth';import Credentials from 'next-auth/providers/credentials';import { authConfig } from './auth.config';import { z } from 'zod';import { sql } from '@vercel/postgres';import type { User } from '@/app/lib/definitions';import bcrypt from 'bcrypt'; async function getUser(email: string): Promise<User | undefined> { try { const user = await sql<User>`SELECT \* FROM users WHERE email=${email}`; return user.rows[0]; } catch (error) { console.error('Failed to fetch user:', error); throw new Error('Failed to fetch user.'); }} export const { auth, signIn, signOut } = NextAuth({ ...authConfig, providers: [ Credentials({ async authorize(credentials) { const parsedCredentials = z .object({ email: z.string().email(), password: z.string().min(6) }) .safeParse(credentials); if (parsedCredentials.success) { const { email, password } = parsedCredentials.data; const user = await getUser(email); if (!user) return null; } return null; }, }), ],});

```

Then, call `bcrypt.compare` to check if the passwords match:

/auth.ts

```

import NextAuth from 'next-auth';import Credentials from 'next-auth/providers/credentials';import { authConfig } from './auth.config';import { sql } from '@vercel/postgres';import { z } from 'zod';import type { User } from '@/app/lib/definitions';import bcrypt from 'bcrypt'; // ... export const { auth, signIn, signOut } = NextAuth({ ...authConfig, providers: [ Credentials({ async authorize(credentials) { // ... if (parsedCredentials.success) { const { email, password } = parsedCredentials.data; const user = await getUser(email); if (!user) return null; const passwordsMatch = await bcrypt.compare(password, user.password); if (passwordsMatch) return user; } console.log('Invalid credentials'); return null; }, }), ],});

```

Finally, if the passwords match you want to return the user, otherwise, return `null` to prevent the user from logging in.

### [Updating the login form](https://nextjs.org/learn/dashboard-app/adding-authentication#updating-the-login-form)

Now you need to connect the auth logic with your login form. In your `actions.ts` file, create a new action called `authenticate`. This action should import the `signIn` function from `auth.ts`:

/app/lib/actions.ts

```

import { signIn } from '@/auth';import { AuthError } from 'next-auth'; // ... export async function authenticate( prevState: string | undefined, formData: FormData,) { try { await signIn('credentials', formData); } catch (error) { if (error instanceof AuthError) { switch (error.type) { case 'CredentialsSignin': return 'Invalid credentials.'; default: return 'Something went wrong.'; } } throw error; }}

```

If there's a `'CredentialsSignin'` error, you want to show an appropriate error message. You can learn about NextAuth.js errors in [the documentation](https://errors.authjs.dev/)

Finally, in your `login-form.tsx` component, you can use React's `useFormState` to call the server action and handle form errors, and use `useFormStatus` to handle the pending state of the form:

app/ui/login-form.tsx

```

'use client'; import { lusitana } from '@/app/ui/fonts';import { AtSymbolIcon, KeyIcon, ExclamationCircleIcon,} from '@heroicons/react/24/outline';import { ArrowRightIcon } from '@heroicons/react/20/solid';import { Button } from '@/app/ui/button';import { useFormState, useFormStatus } from 'react-dom';import { authenticate } from '@/app/lib/actions'; export default function LoginForm() { const [errorMessage, dispatch] = useFormState(authenticate, undefined); return ( <form action={dispatch} className="space-y-3"> <div className="flex-1 rounded-lg bg-gray-50 px-6 pb-4 pt-8"> <h1 className={`${lusitana.className} mb-3 text-2xl`}> Please log in to continue. </h1> <div className="w-full"> <div> <label className="mb-3 mt-5 block text-xs font-medium text-gray-900" htmlFor="email" > Email </label> <div className="relative"> <input className="peer block w-full rounded-md border border-gray-200 py-[9px] pl-10 text-sm outline-2 placeholder:text-gray-500" id="email" type="email" name="email" placeholder="Enter your email address" required /> <AtSymbolIcon className="pointer-events-none absolute left-3 top-1/2 h-[18px] w-[18px] -translate-y-1/2 text-gray-500 peer-focus:text-gray-900" /> </div> </div> <div className="mt-4"> <label className="mb-3 mt-5 block text-xs font-medium text-gray-900" htmlFor="password" > Password </label> <div className="relative"> <input className="peer block w-full rounded-md border border-gray-200 py-[9px] pl-10 text-sm outline-2 placeholder:text-gray-500" id="password" type="password" name="password" placeholder="Enter password" required minLength={6} /> <KeyIcon className="pointer-events-none absolute left-3 top-1/2 h-[18px] w-[18px] -translate-y-1/2 text-gray-500 peer-focus:text-gray-900" /> </div> </div> </div> <LoginButton /> <div className="flex h-8 items-end space-x-1" aria-live="polite" aria-atomic="true" > {errorMessage && ( <> <ExclamationCircleIcon className="h-5 w-5 text-red-500" /> <p className="text-sm text-red-500">{errorMessage}</p> </> )} </div> </div> </form> );} function LoginButton() { const { pending } = useFormStatus(); return ( <Button className="mt-4 w-full" aria-disabled={pending}> Log in <ArrowRightIcon className="ml-auto h-5 w-5 text-gray-50" /> </Button> );}

```

## [Adding the logout functionality](https://nextjs.org/learn/dashboard-app/adding-authentication#adding-the-logout-functionality)

To add the logout functionality to `<SideNav />`, call the `signOut` function from `auth.ts` in your `<form>` element:

/ui/dashboard/sidenav.tsx

```

import Link from 'next/link';import NavLinks from '@/app/ui/dashboard/nav-links';import AcmeLogo from '@/app/ui/acme-logo';import { PowerIcon } from '@heroicons/react/24/outline';import { signOut } from '@/auth'; export default function SideNav() { return ( <div className="flex h-full flex-col px-3 py-4 md:px-2"> // ... <div className="flex grow flex-row justify-between space-x-2 md:flex-col md:space-x-0 md:space-y-2"> <NavLinks /> <div className="hidden h-auto w-full grow rounded-md bg-gray-50 md:block"></div> <form action={async () => { 'use server'; await signOut(); }} > <button className="flex h-[48px] grow items-center justify-center gap-2 rounded-md bg-gray-50 p-3 text-sm font-medium hover:bg-sky-100 hover:text-blue-600 md:flex-none md:justify-start md:p-2 md:px-3"> <PowerIcon className="w-6" /> <div className="hidden md:block">Sign Out</div> </button> </form> </div> </div> );}

```

## [Try it out](https://nextjs.org/learn/dashboard-app/adding-authentication#try-it-out)

Now, try it out. You should be able to log in and out of your application using the following credentials:

- Email: `user@nextmail.com`

- Password: `123456`

# Adding Metadata

Metadata is crucial for SEO and shareability. In this chapter, we'll discuss how you can add metadata to your Next.js application.

In this chapter...

Here are the topics we’ll cover

What metadata is.

Types of metadata.

How to add an Open Graph image using metadata.

How to add a favicon using metadata.

## [What is metadata?](https://nextjs.org/learn/dashboard-app/adding-metadata#what-is-metadata)

In web development, metadata provides additional details about a webpage. Metadata is not visible to the users visiting the page. Instead, it works behind the scenes, embedded within the page's HTML, usually within the `<head>` element. This hidden information is crucial for search engines and other systems that need to understand your webpage's content better.

## [Why is metadata important?](https://nextjs.org/learn/dashboard-app/adding-metadata#why-is-metadata-important)

Metadata plays a significant role in enhancing a webpage's SEO, making it more accessible and understandable for search engines and social media platforms. Proper metadata helps search engines effectively index webpages, improving their ranking in search results. Additionally, metadata like Open Graph improves the appearance of shared links on social media, making the content more appealing and informative for users.

## [Types of metadata](https://nextjs.org/learn/dashboard-app/adding-metadata#types-of-metadata)

There are various types of metadata, each serving a unique purpose. Some common types include:

\*\*Title Metadata\*\*: Responsible for the title of a webpage that is displayed on the browser tab. It's crucial for SEO as it helps search engines understand what the webpage is about.

```

<title>Page Title</title>

```

\*\*Description Metadata\*\*: This metadata provides a brief overview of the webpage content and is often displayed in search engine results.

```

<meta name="description" content="A brief description of the page content." />

```

\*\*Keyword Metadata\*\*: This metadata includes the keywords related to the webpage content, helping search engines index the page.

```

<meta name="keywords" content="keyword1, keyword2, keyword3" />

```

\*\*Open Graph Metadata\*\*: This metadata enhances the way a webpage is represented when shared on social media platforms, providing information such as the title, description, and preview image.

```

<meta property="og:title" content="Title Here" /><meta property="og:description" content="Description Here" /><meta property="og:image" content="image\_url\_here" />

```

\*\*Favicon Metadata\*\*: This metadata links the favicon (a small icon) to the webpage, displayed in the browser's address bar or tab.

```

<link rel="icon" href="path/to/favicon.ico" />

```

## [Adding metadata](https://nextjs.org/learn/dashboard-app/adding-metadata#adding-metadata)

Next.js has a Metadata API that can be used to define your application metadata. There are two ways you can add metadata to your application:

- \*\*Config-based\*\*: Export a [static `metadata` object](https://nextjs.org/docs/app/api-reference/functions/generate-metadata#metadata-object) or a dynamic [`generateMetadata` function](https://nextjs.org/docs/app/api-reference/functions/generate-metadata#generatemetadata-function) in a `layout.js` or `page.js` file.

- \*\*File-based\*\*: Next.js has a range of special files that are specifically used for metadata purposes:

- `favicon.ico`, `apple-icon.jpg`, and `icon.jpg`: Utilized for favicons and icons

- `opengraph-image.jpg` and `twitter-image.jpg`: Employed for social media images

- `robots.txt`: Provides instructions for search engine crawling

- `sitemap.xml`: Offers information about the website's structure

You have the flexibility to use these files for static metadata, or you can generate them programmatically within your project.

With both these options, Next.js will automatically generate the relevant `<head>` elements for your pages.

### [Favicon and Open Graph image](https://nextjs.org/learn/dashboard-app/adding-metadata#favicon-and-open-graph-image)

In your `/public` folder, you'll notice you have two images: `favicon.ico` and `opengraph-image.jpg`.

Move these images to the root of your `/app` folder.

After doing this, Next.js will automatically identify and use these files as your favicon and OG image. You can verify this by checking the `<head>` element of your application in dev tools.

> \*\*Good to know:\*\* You can also create dynamic OG images using the [`ImageResponse`](https://nextjs.org/docs/app/api-reference/functions/image-response) constructor.

### [Page title and descriptions](https://nextjs.org/learn/dashboard-app/adding-metadata#page-title-and-descriptions)

You can also include a [`metadata` object](https://nextjs.org/docs/app/api-reference/functions/generate-metadata#metadata-fields) from any `layout.js` or `page.js` file to add additional page information like title and description. Any metadata in `layout.js` will be inherited by all pages that use it.

In your root layout, create a new `metadata` object with the following fields:

/app/layout.tsx

```

import { Metadata } from 'next'; export const metadata: Metadata = { title: 'Acme Dashboard', description: 'The official Next.js Course Dashboard, built with App Router.', metadataBase: new URL('https://next-learn-dashboard.vercel.sh'),}; export default function RootLayout() { // ...}

```

Next.js will automatically add the title and metadata to your application.

But what if you want to add a custom title for a specific page? You can do this by adding a `metadata` object to the page itself. Metadata in nested pages will override the metadata in the parent.

For example, in the `/dashboard/invoices` page, you can update the page title:

/app/dashboard/invoices/page.tsx

```

import { Metadata } from 'next'; export const metadata: Metadata = { title: 'Invoices | Acme Dashboard',};

```

This works, but we are repeating the title of the application in every page. If something changes, like the company name, you'd have to update it on every page.

Instead, you can use the `title.template` field in the `metadata` object to define a template for your page titles. This template can include the page title, and any other information you want to include.

In your root layout, update the `metadata` object to include a template:

/app/layout.tsx

```

import { Metadata } from 'next'; export const metadata: Metadata = { title: { template: '%s | Acme Dashboard', default: 'Acme Dashboard', }, description: 'The official Next.js Learn Dashboard built with App Router.', metadataBase: new URL('https://next-learn-dashboard.vercel.sh'),};

```

The `%s` in the template will be replaced with the specific page title.

Now, in your `/dashboard/invoices` page, you can add the page title:

/app/dashboard/invoices/page.tsx

```

export const metadata: Metadata = { title: 'Invoices',};

```

Navigate to the `/dashboard/invoices` page and check the `<head>` element. You should see the page title is now `Invoices | Acme Dashboard`.

## [Practice: Adding metadata](https://nextjs.org/learn/dashboard-app/adding-metadata#practice-adding-metadata)

Now that you've learned about metadata, practice by adding titles to your other pages:

1. `/login` page.

2. `/dashboard/` page.

3. `/dashboard/customers` page.

4. `/dashboard/invoices/create` page.

5. `/dashboard/invoices/[id]/edit` page.

The Next.js Metadata API is powerful and flexible, giving you full control over your application's metadata. Here, we've shown you how to add some basic metadata, but you can add multiple fields, including `keywords`, `robots`, `canonical`, and more. Feel free to explore the [documentation](https://nextjs.org/docs/app/api-reference/functions/generate-metadata), and add any additional metadata you want to your application.