Curso de Next.js 14

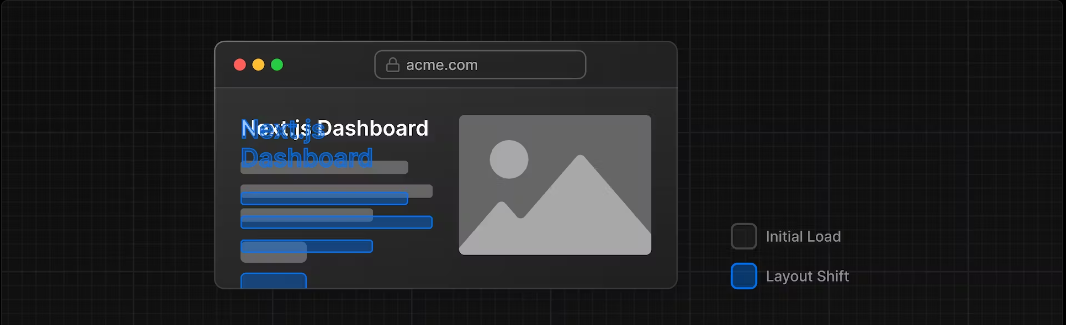
1) Css Styling

2) Optimizing Fonts and 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 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.



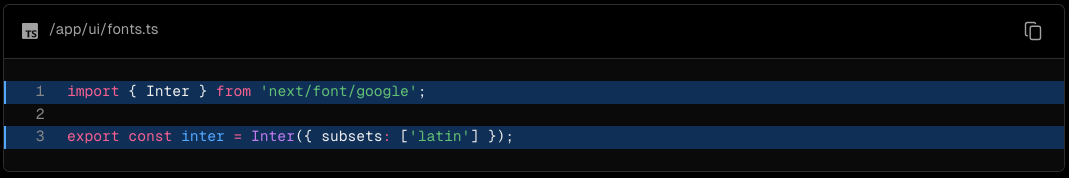
Next.js automatically optimizes fonts in the application when you use the next/font module. It downloads font files at build time and hosts them with your other static assets. This means when a user visits your application, there are no additional network requests for fonts which would impact performance.

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 you'd like to load. In this case, 'latin':



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



By adding Inter to the <body> element, the font will be applied throughout your application. Here, you're also adding the Tailwind antialiased 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

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.

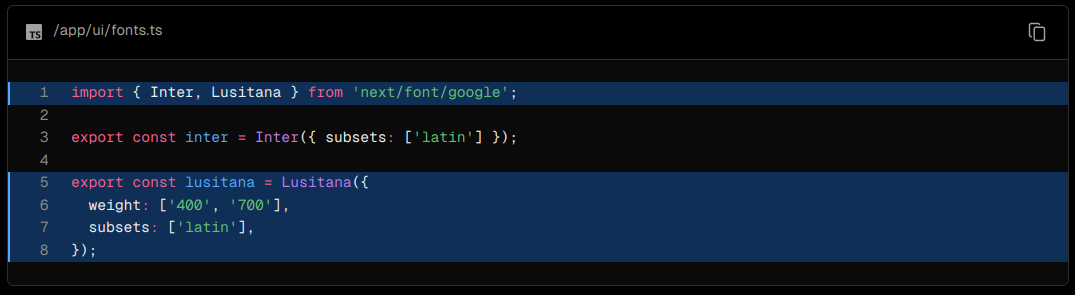
Once you're ready, expand the code snippet below to see the solution.

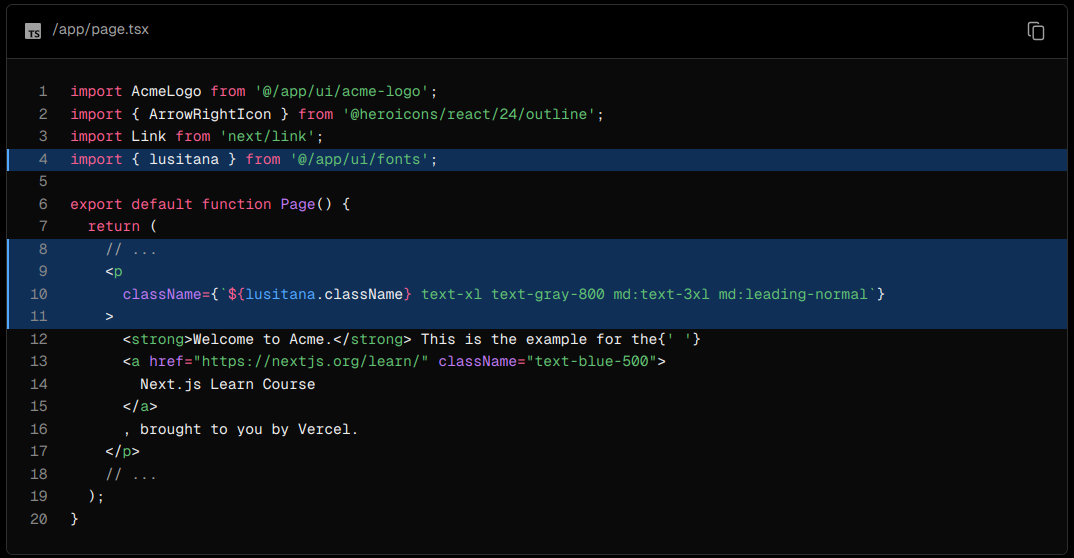
Hints:

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

Visit the Google Fonts website and search for Lusitana to see what options are available.

See the documentation for adding multiple fonts and the full list of options.

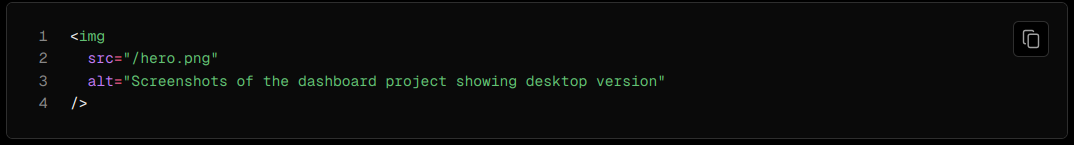




Why optimize images?

Next.js can serve static assets, like images, under the top-level /public folder. Files inside /public can be referenced in your application.

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



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

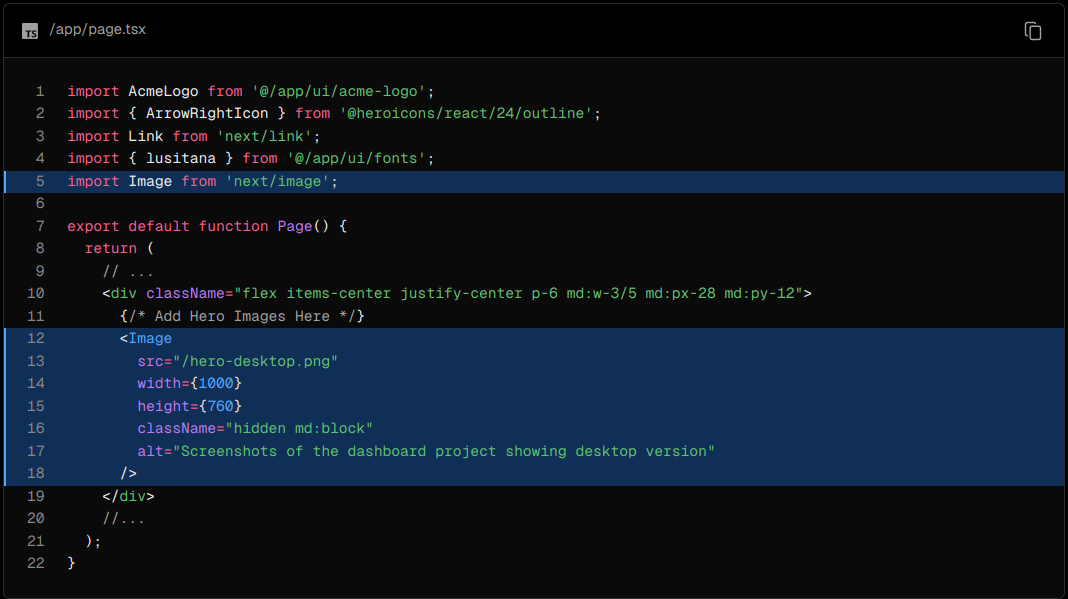
The <Image> Component is an extension of the HTML <img> tag, and comes with automatic image optimization, such as:

* 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 and AVIF, when the browser supports it.

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.

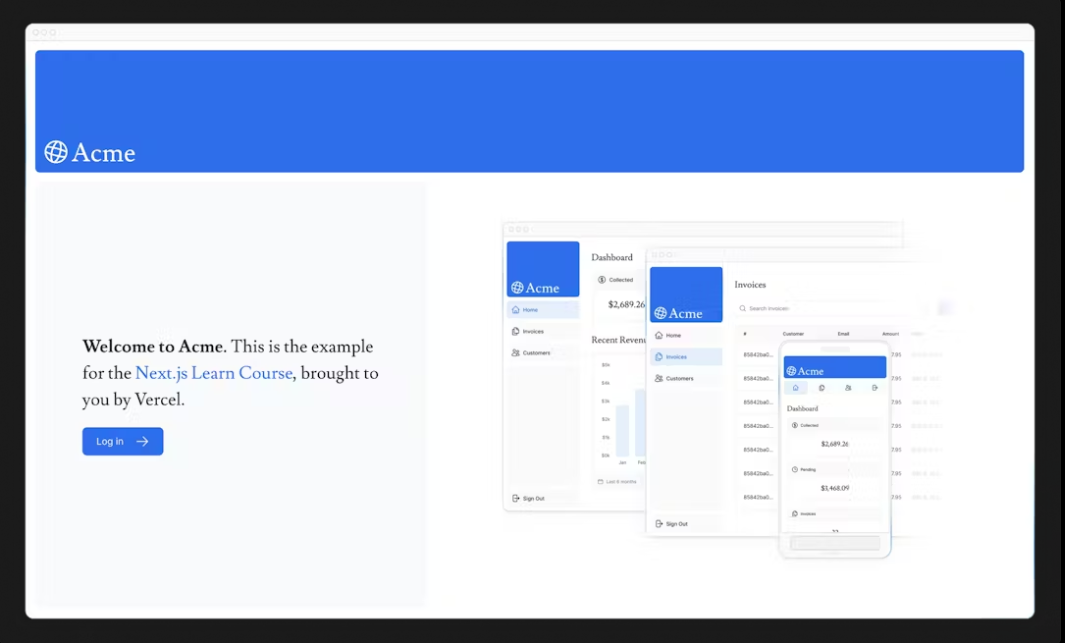
In your /app/page.tsx file, import the component from next/image. Then, add the image under the comment:



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:



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.

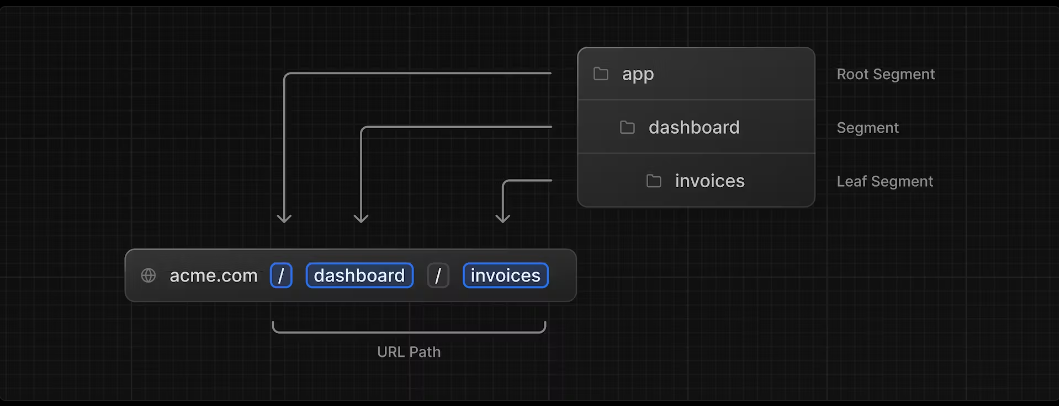
* The image should have a width of 560 and height of 620 pixels.
* It should be shown on mobile screens, and hidden on desktop - you can use dev tools to check if the desktop and mobile images are swapped correctly.

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

4) Creating Layouts and Pages

Nested routing

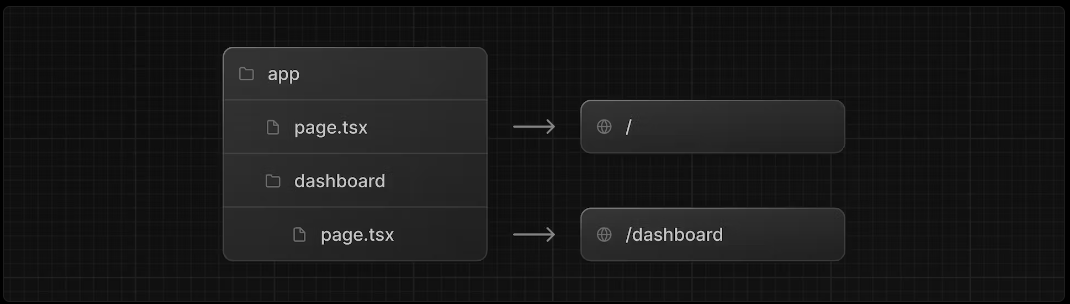
Next.js uses file-system routing where folders are used to create nested routes. Each folder represents a route segment that maps to a URL segment.



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:



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

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:



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

Practice: Creating the dashboard pages

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

Customers Page: The page should be accessible on http://localhost:3000/dashboard/customers. For now, it should return a <p>Customers Page</p> element.

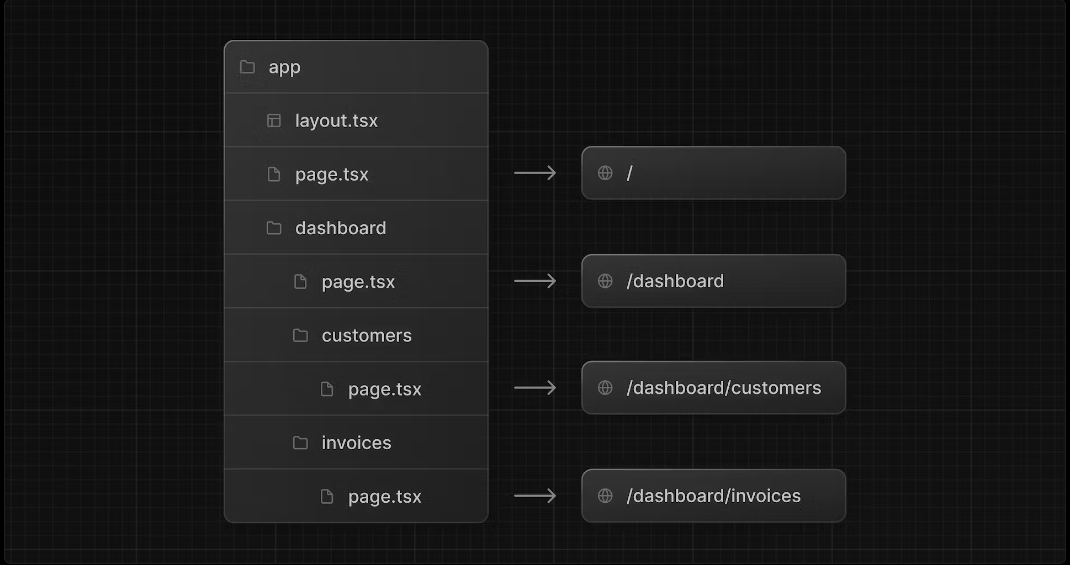
Invoices Page: The invoices page should be accessible on http://localhost:3000/dashboard/invoices. For now, also return a <p>Invoices Page</p> element.

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

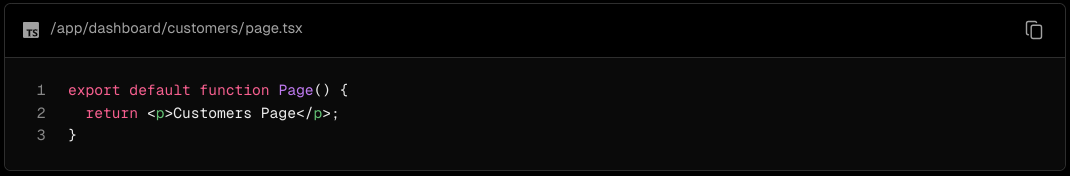
This is how you can create different pages in Next.js: create a new route segment using a folder, and add a page file inside it.

By having a special name for page files, Next.js allows you to colocate 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.

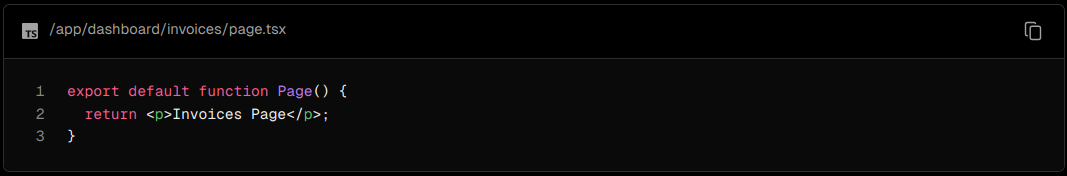
You should have the following folder structure:



Customers Page:



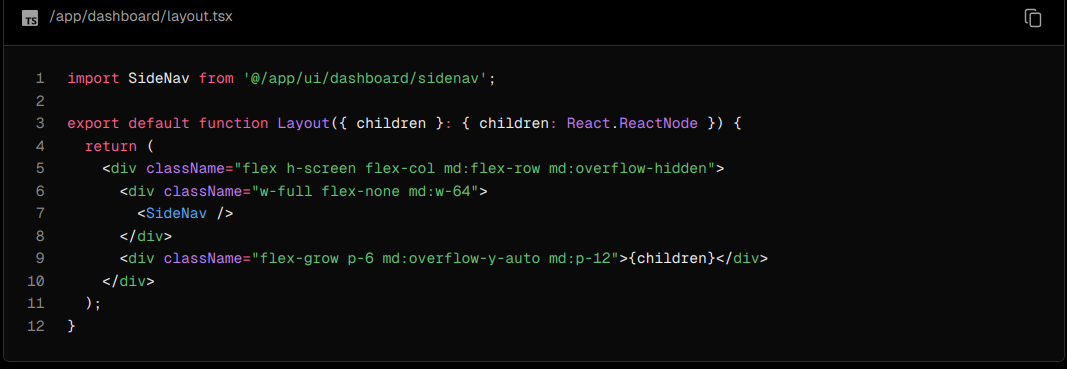
Invoices Page:



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!

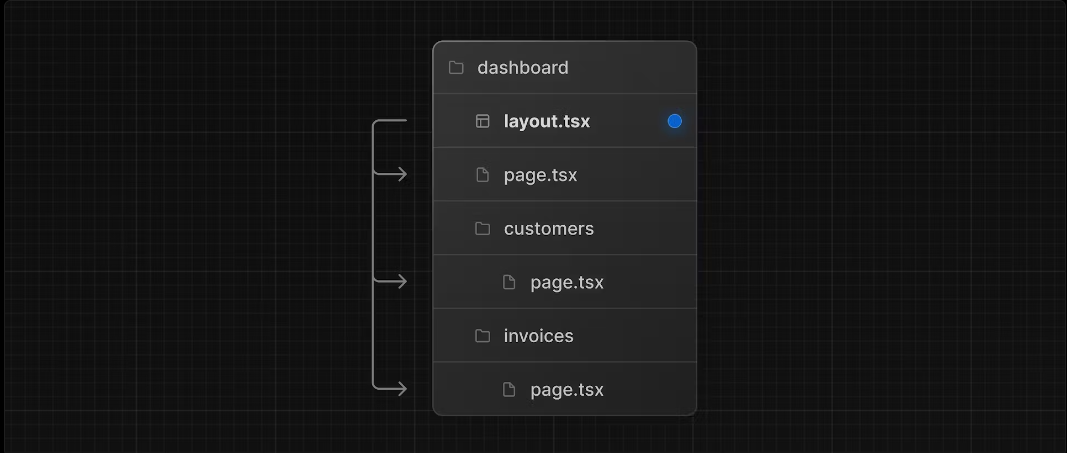
Inside the /dashboard folder, add a new file called layout.tsx and paste the following code:



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.

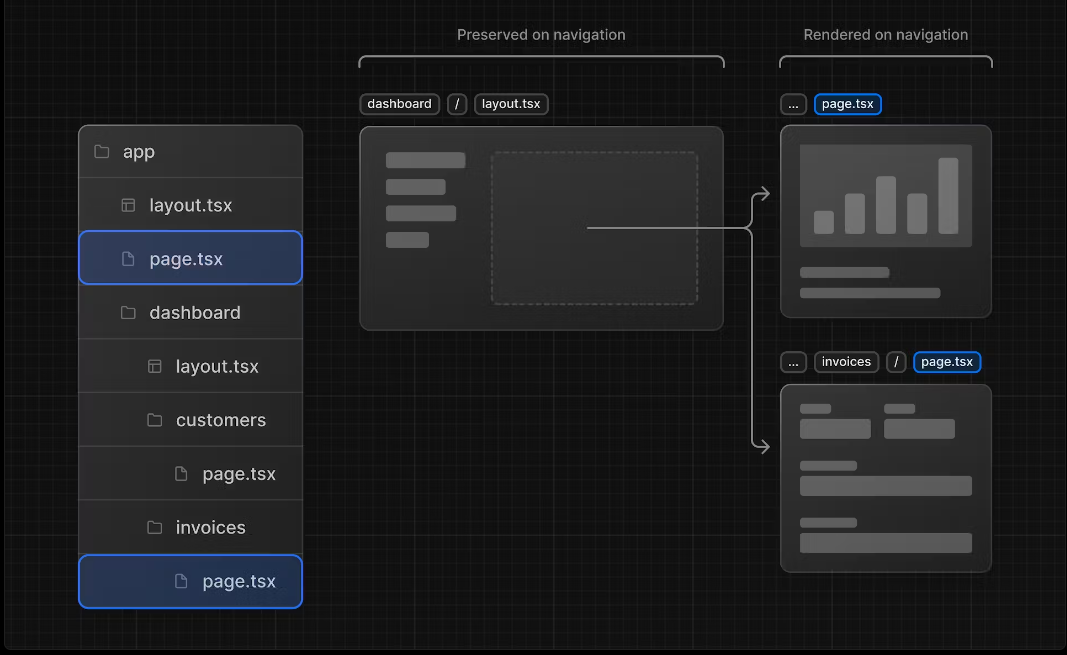
The <Layout /> component receives a children prop. This child can either be a page or another layout. In your case, the pages inside /dashboard will automatically be nested inside a <Layout /> like so:



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



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:



Root layout

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



This is called a root layout 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).

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.

5) 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.

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.

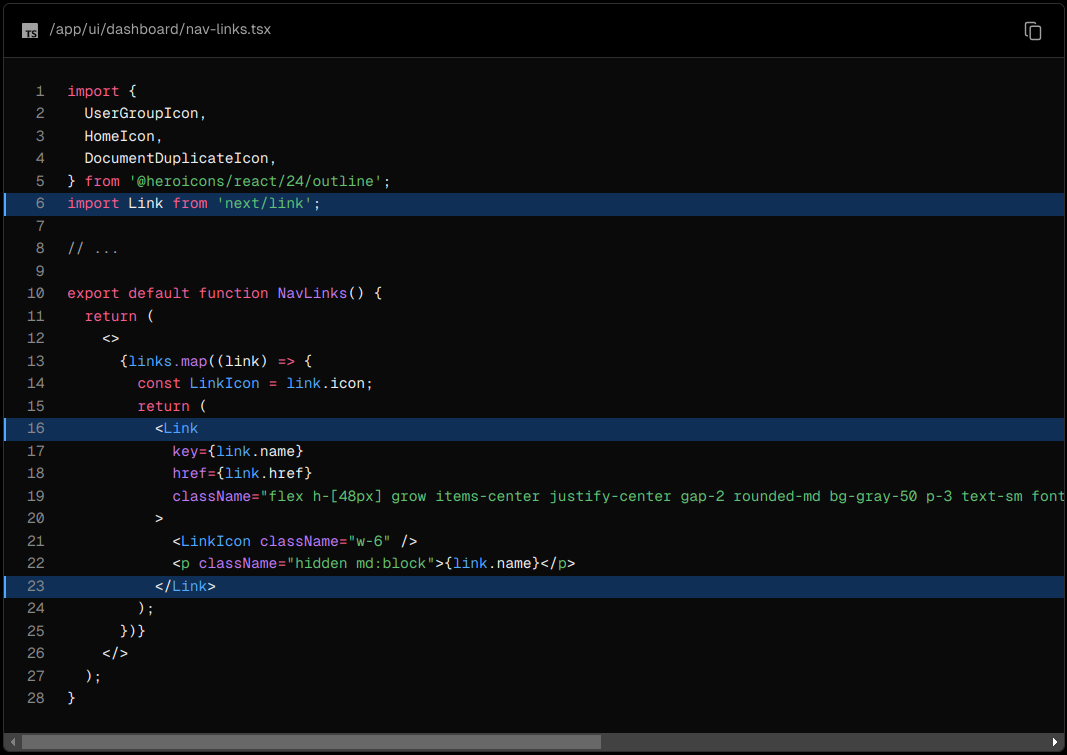
Did you see it?

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

The <Link> component

In Next.js, you can use the <Link /> Component to link between pages in your application. <Link> allows you to do client-side navigation with JavaScript.

To use the <Link /> component, open /app/ui/dashboard/nav-links.tsx, and import the Link component from next/link. Then, replace the <a> tag with <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

To improve the navigation experience, Next.js automatically code splits your application by route segments. This is different from a traditional React 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> 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.

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() that you can use to check the path and implement this pattern.

Since usePathname() 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:

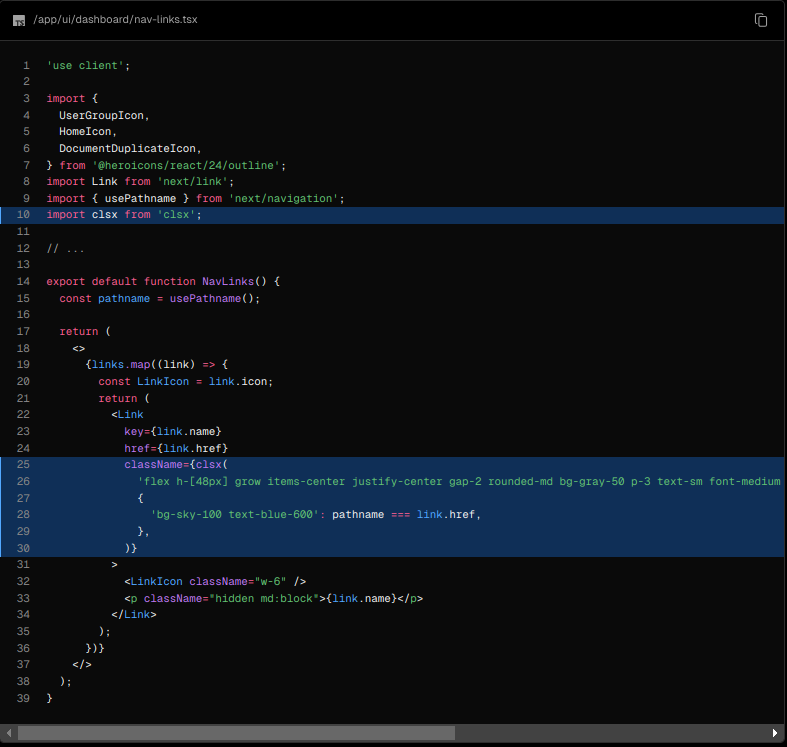


Next, assign the path to a variable called pathname inside your <NavLinks /> component:



You can use the clsx library introduced in the chapter on 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:



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

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!

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.

Good to know:

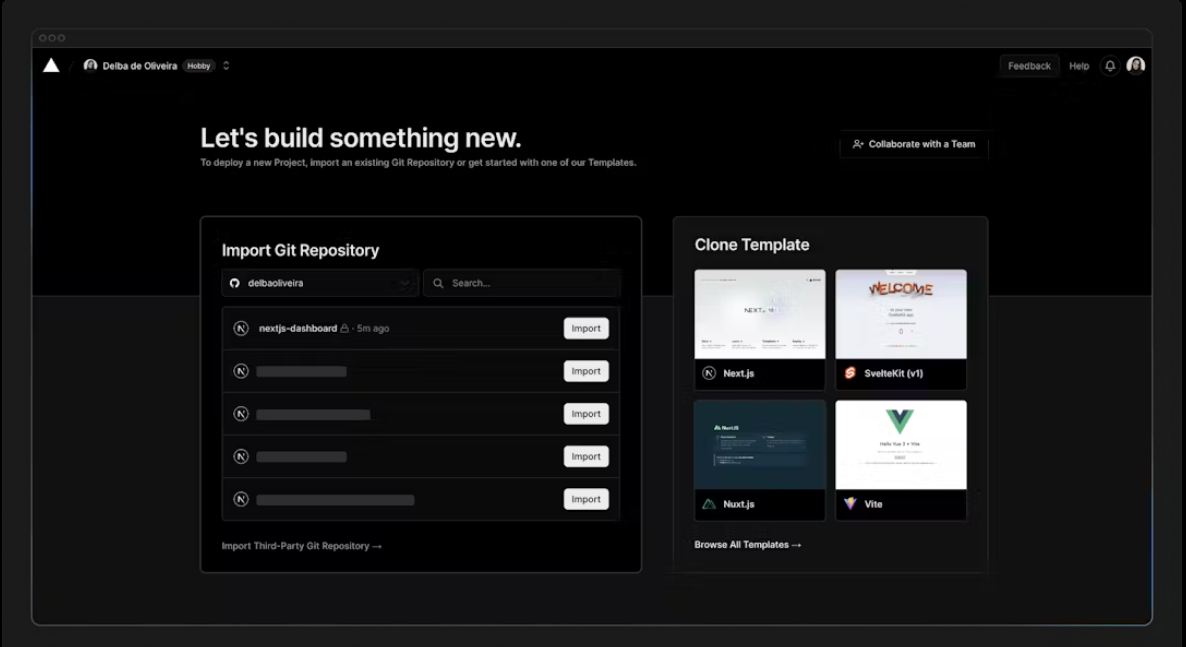
* You can also use other Git provider like GitLab or Bitbucket.
* If you're new to GitHub, we recommend the GitHub Desktop App for a simplified development workflow.

Create a Vercel account

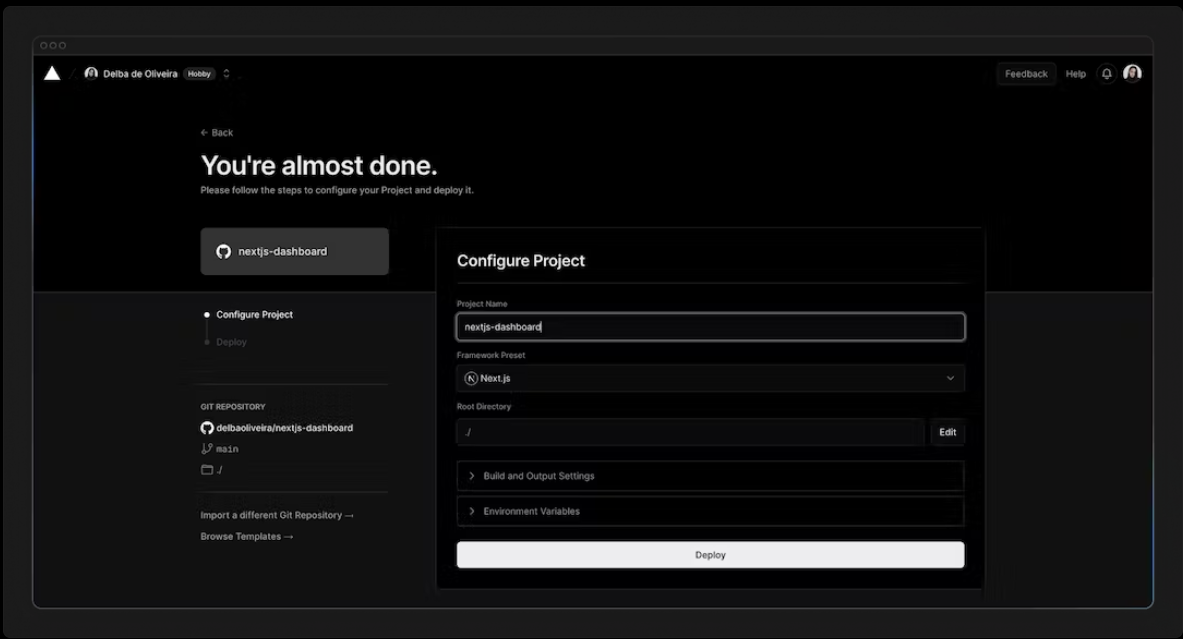
Visit vercel.com/signup to create an account. Choose the free "hobby" plan. Select Continue with GitHub to connect your GitHub and Vercel accounts.

Connect and deploy your project

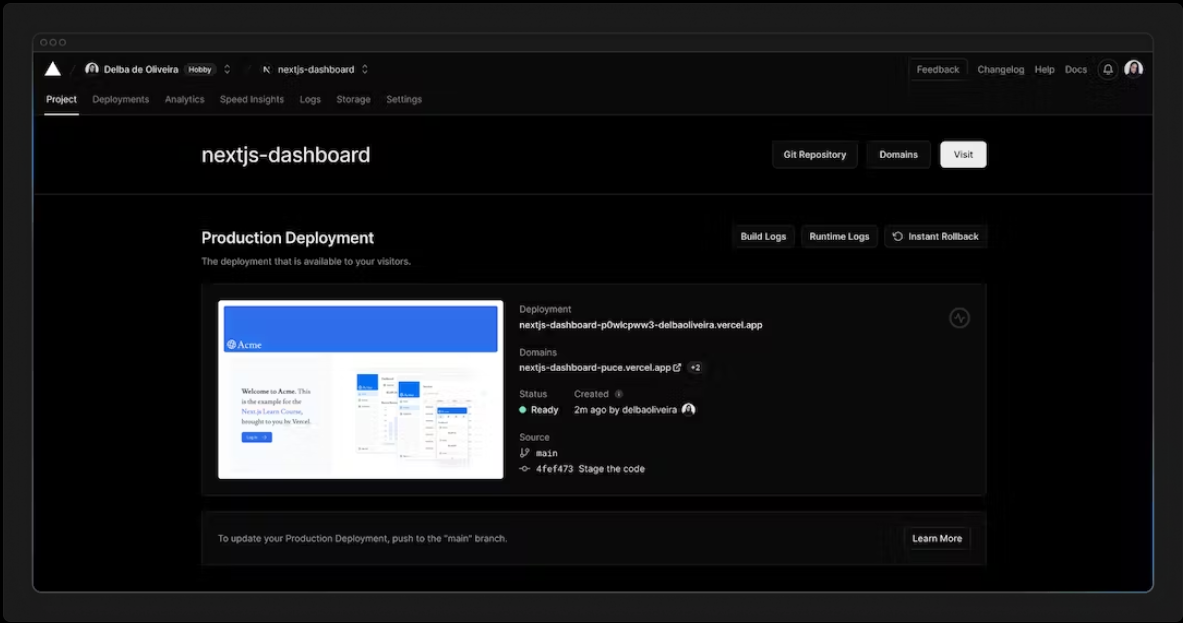
Next, you'll be taken to this screen where you can select and import the GitHub repository you've just created:



Name your project and click Deploy.



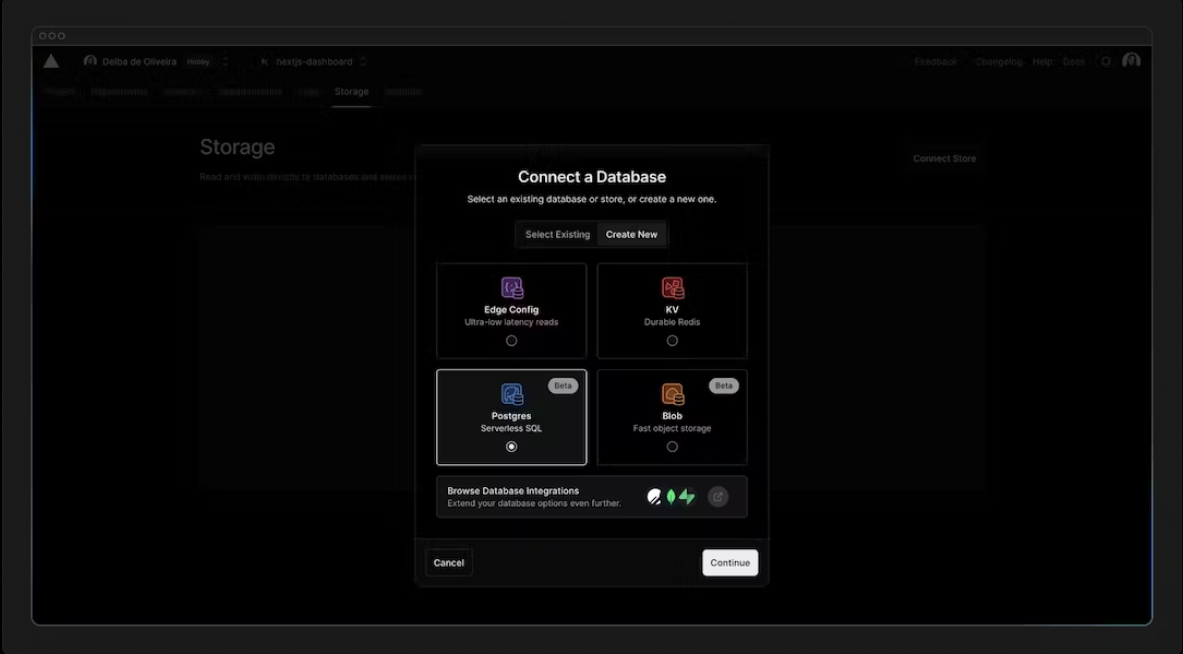
Hooray! 🎉 Your project is now deployed.



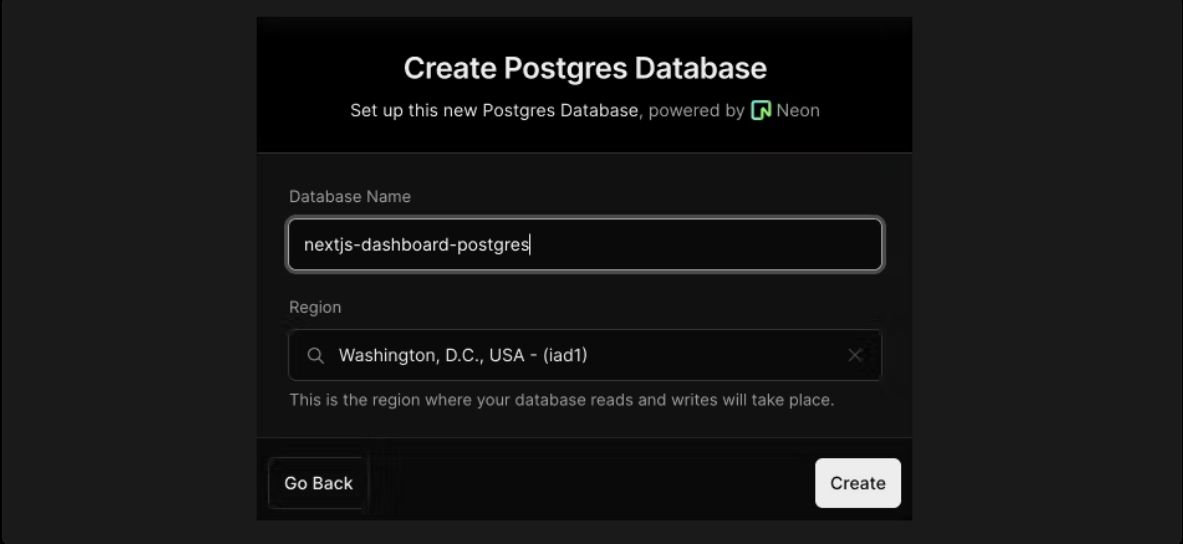
By connecting your GitHub repository, whenever you push changes to your main branch, Vercel will automatically redeploy your application with no configuration needed. When opening pull requests, you'll also have instant previews which allow you to catch deployment errors early and share a preview of your project with team members for feedback.

Create a Postgres database

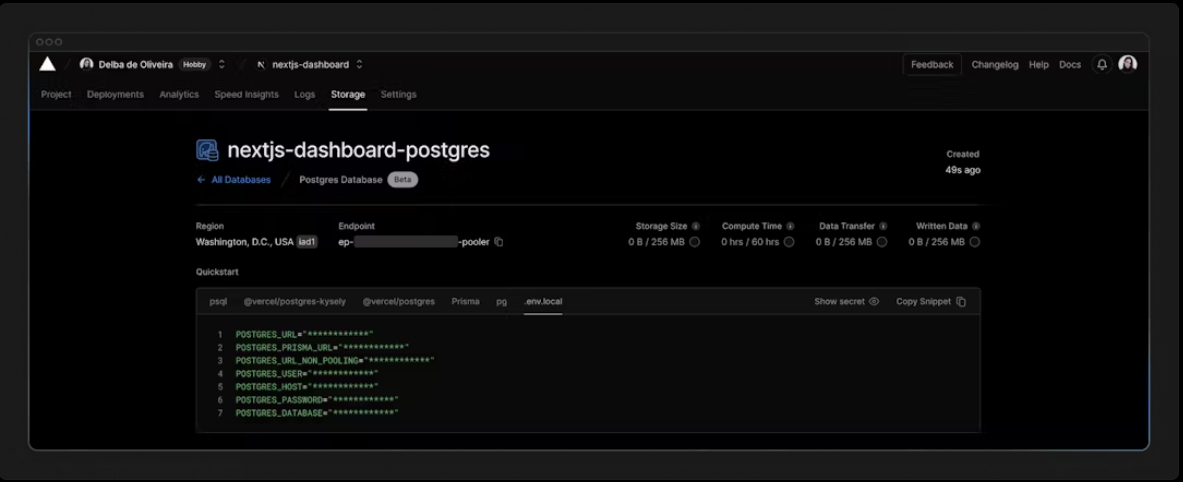
Next, to set up a database, click Continue to Dashboard and select the Storage tab from your project dashboard. Select Connect Store → Create New → Postgres → Continue.



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 for all new Vercel projects. By placing your database in the same region or close to your application code, you can reduce latency for data requests.



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



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.

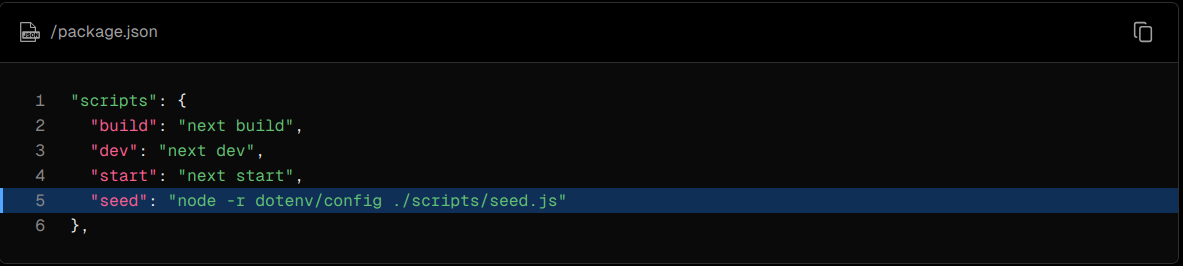
Seed your database

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:



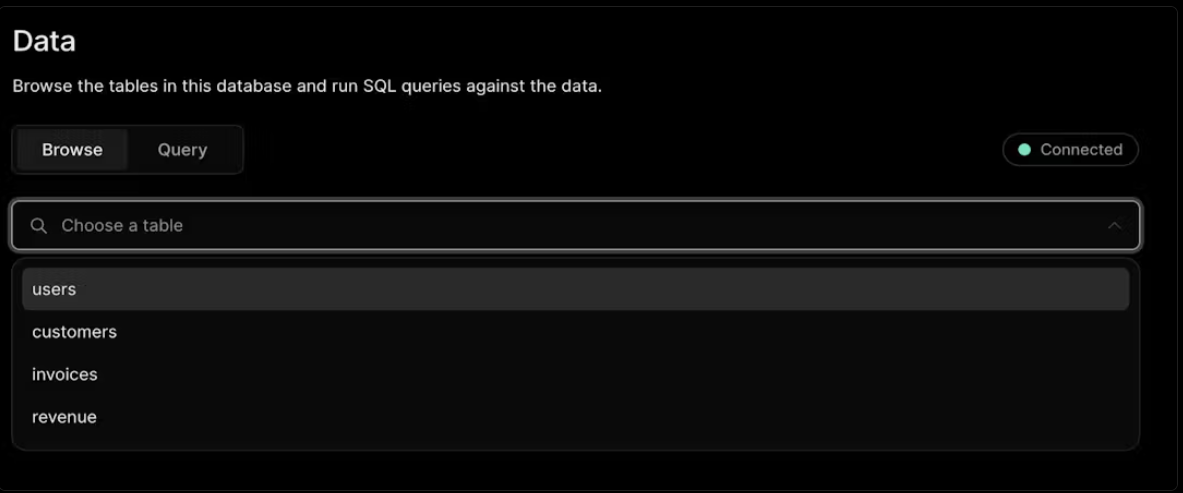
This is the command that will execute seed.js.

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

Exploring your database

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

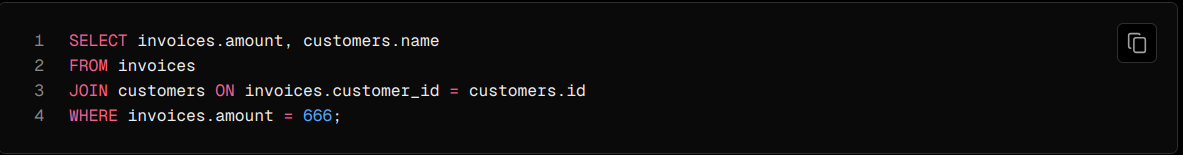
In this section, you'll find the four new tables: users, customers, invoices, and revenue.

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

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:



7) 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.

Choosing how to fetch 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.

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 like Postgres, you can do this with SQL, or an ORM like Prisma.

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.

Let's learn more about React Server Components.

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.

Using SQL

For your dashboard project, you'll write database queries using the 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.

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 function from @vercel/postgres. This function allows you to query your database:



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.

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:



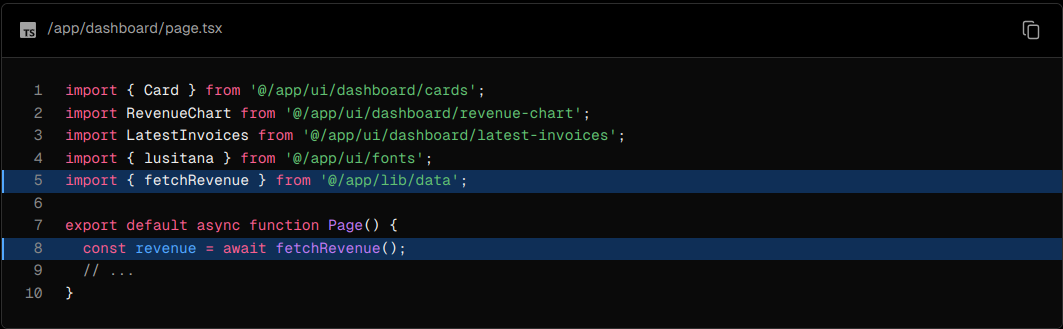
In the code above:

Page is an async component. This allows you to use await to fetch data.

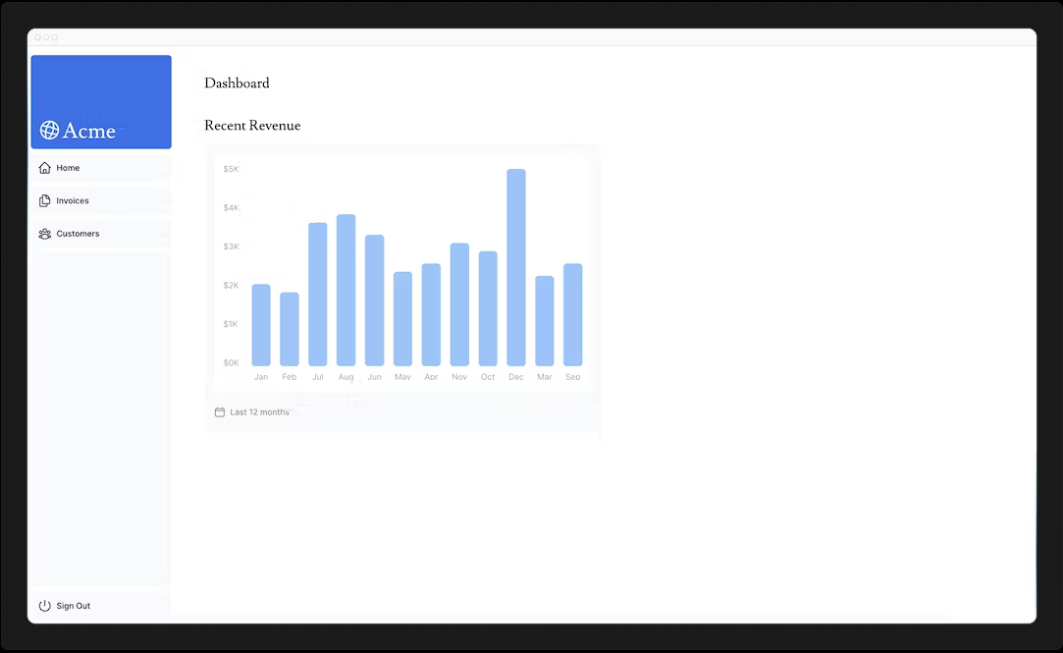
There are also 3 components which receive data: <Card>, <RevenueChart>, and <LatestInvoices>. They are currently commented out to prevent the application from erroring.

Fetching data for <RevenueChart/>

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



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.



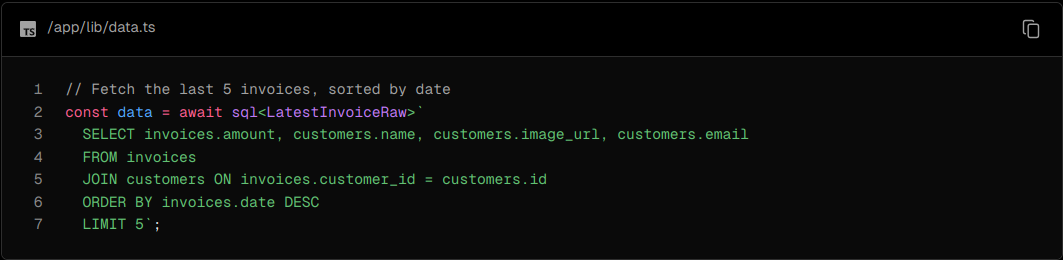
Let's continue importing some more data queries!

Fetching data for <LatestInvoices/>

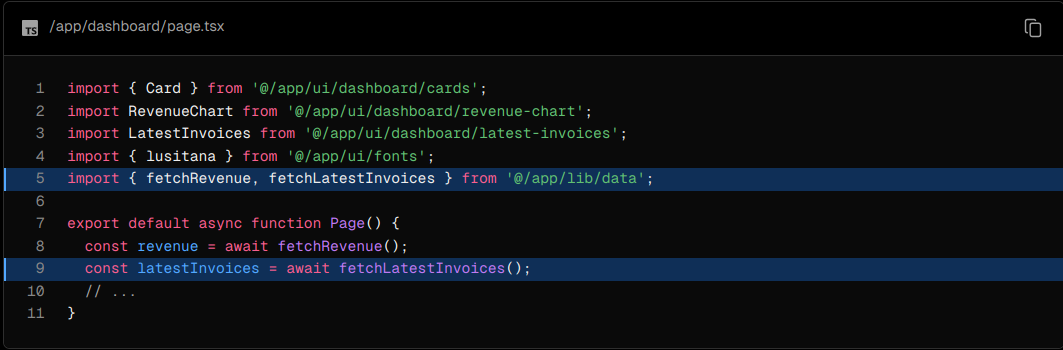
For the <LatestInvoices /> component, we need to get the latest 5 invoices, sorted by date.

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:

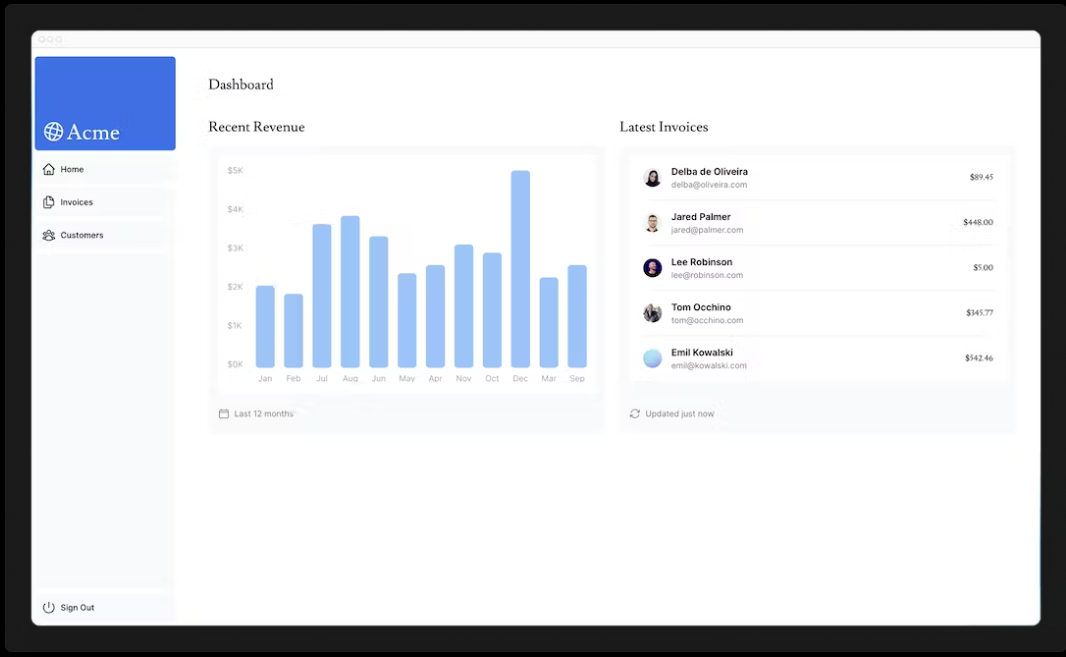


In your page, import the fetchLatestInvoices function:



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!



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:

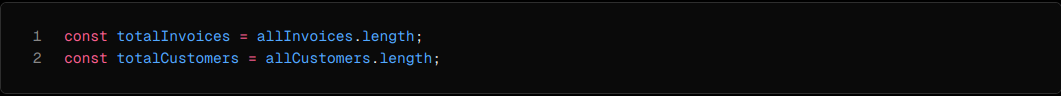
Total amount of invoices collected.

Total amount of invoices pending.

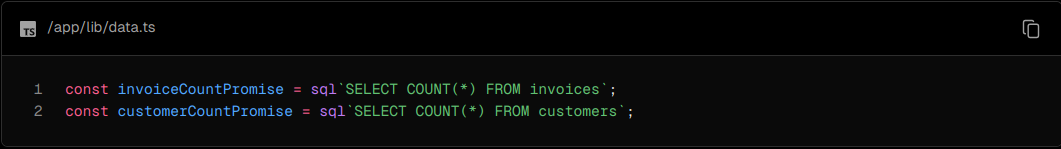
Total number of invoices.

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:



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:

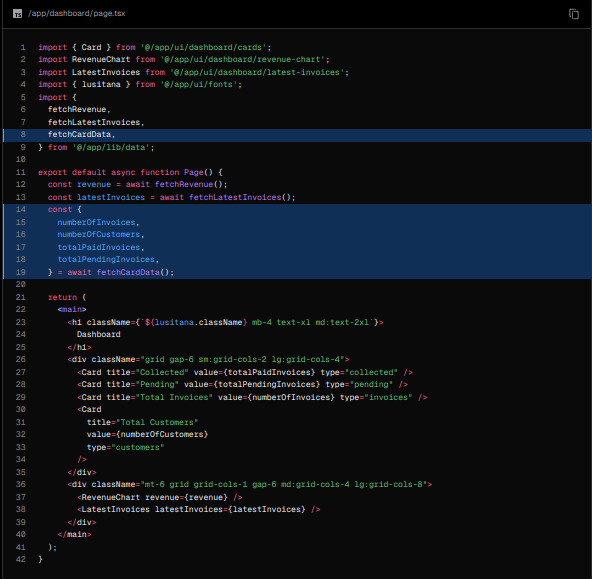


The function you will need to import is called fetchCardData. You will need to destructure the values returned from the function.

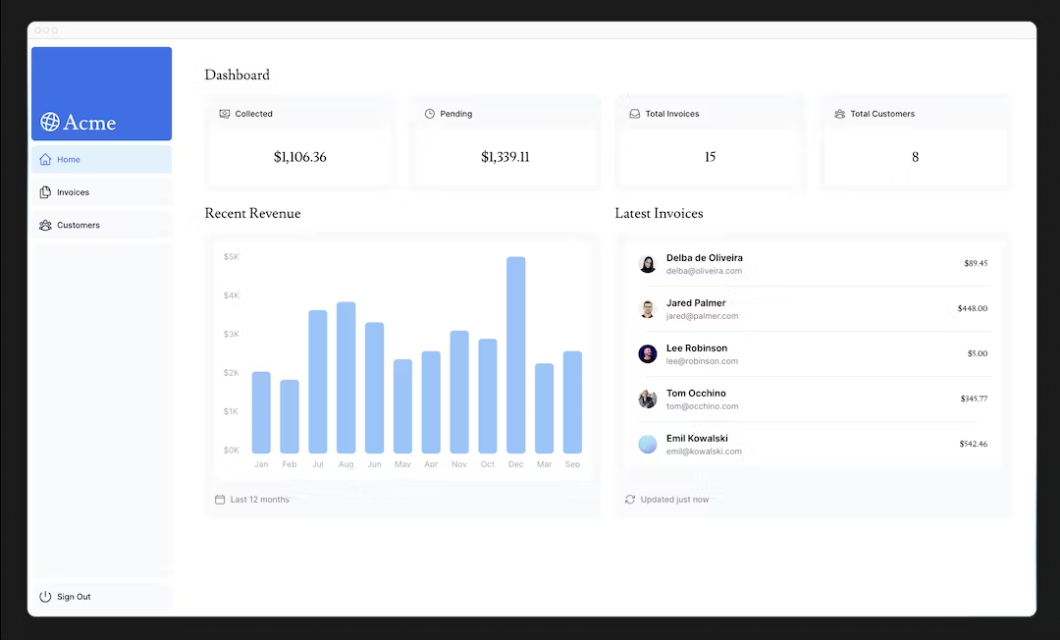
Hint:

* 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:



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



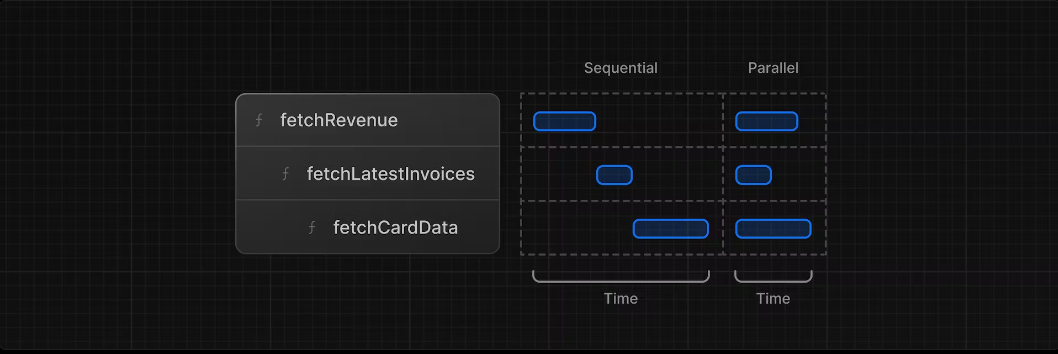
However... there are two things you need to be aware of:

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?

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.



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



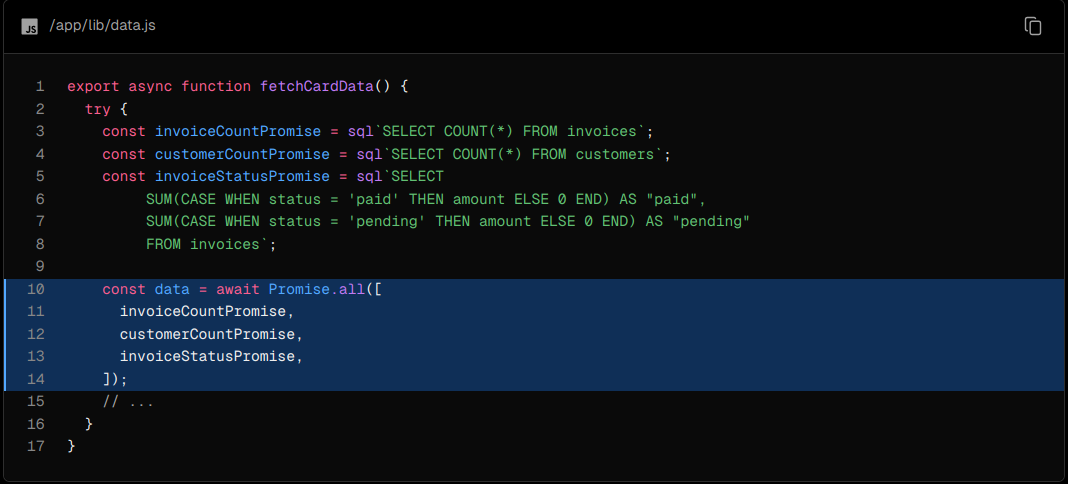
This pattern is not necessarily bad. There may be cases where you want waterfalls because you want a condition to be satisfied before you make the next request. For example, you might want to fetch a user's ID and profile information first. Once you have the ID, you might then proceed to fetch their 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.

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() or 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:



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?

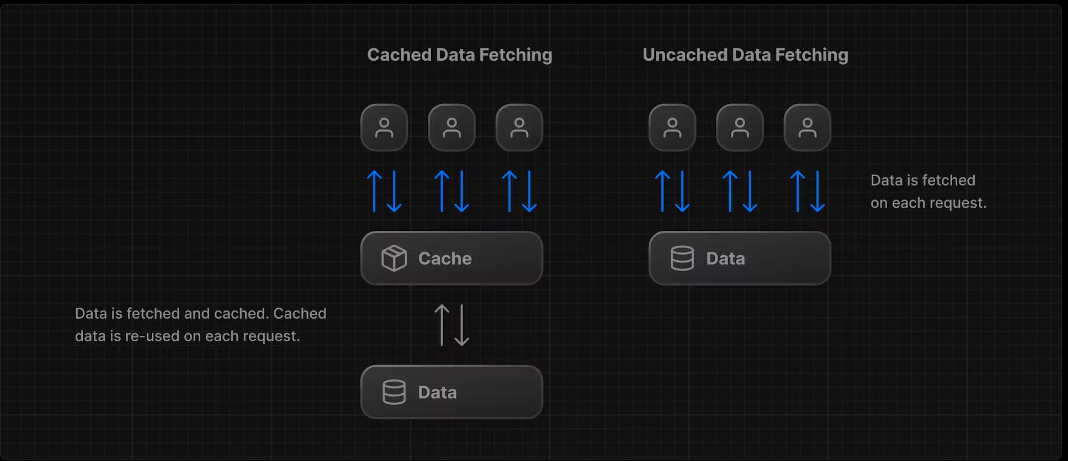
8) Static and Dynamic Rendering

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.

What is Static Rendering?

With static rendering, data fetching and rendering happens on the server at build time (when you deploy) or during revalidation. The result can then be distributed and cached in a Content Delivery Network (CDN).



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.

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:

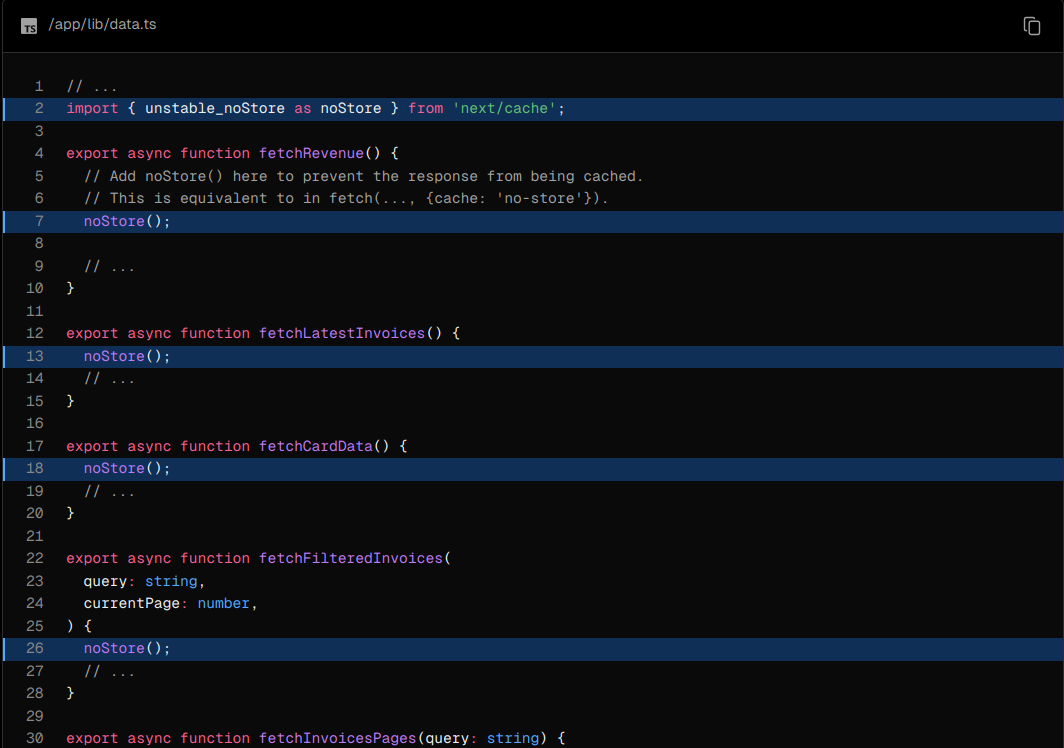
* Real-Time Data - Dynamic rendering allows your application to display real-time or frequently updated data. This is ideal for applications where data changes often.
* User-Specific Content - It's easier to serve personalized content, such as dashboards or user profiles, and update the data based on 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.

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:

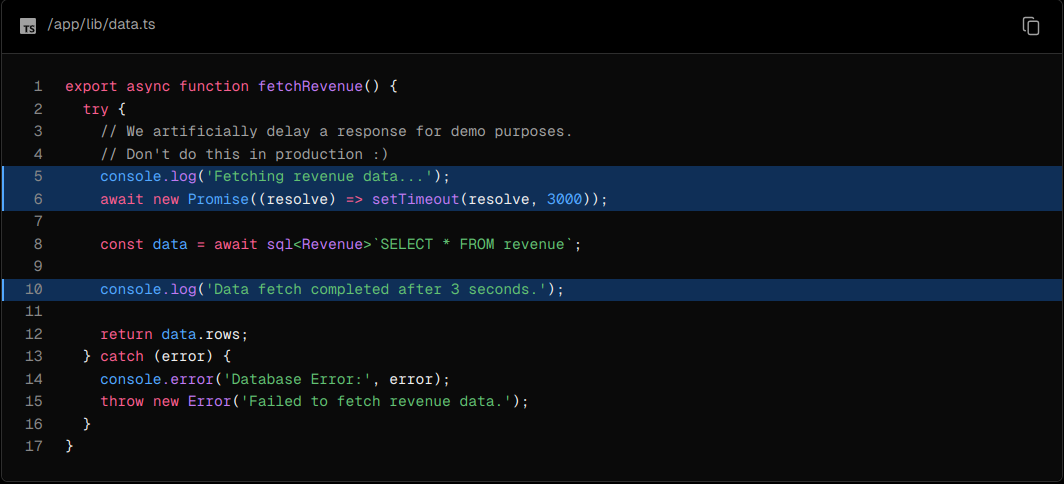


Note: unstable\_noStore is an experimental API and may change in the future. If you prefer to use a stable API in your own projects, you can also use the Segment Config Option export const dynamic = "force-dynamic".

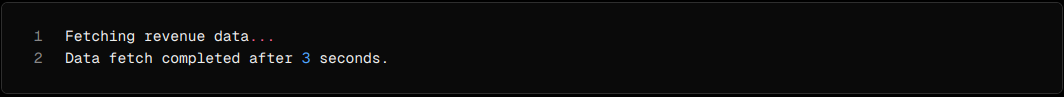
Simulating a Slow Data Fetch

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():



Now open 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:



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:

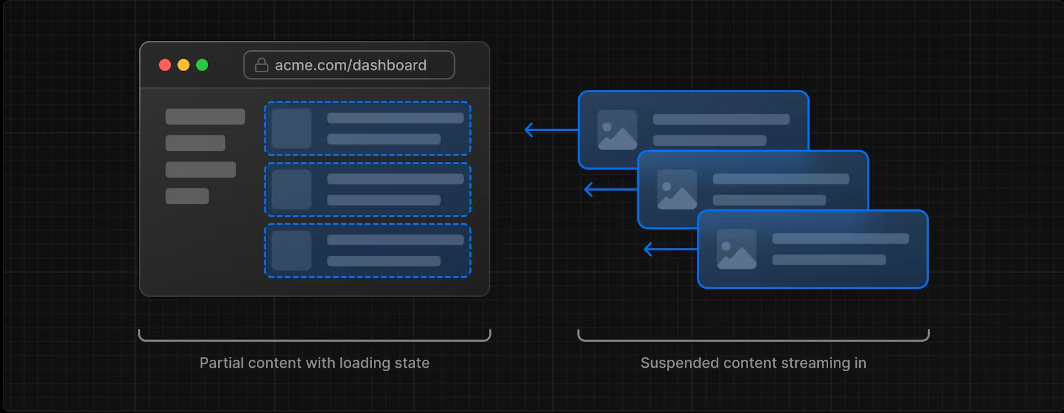
With dynamic rendering, your application is only as fast as your slowest data fetch.

9) Streaming

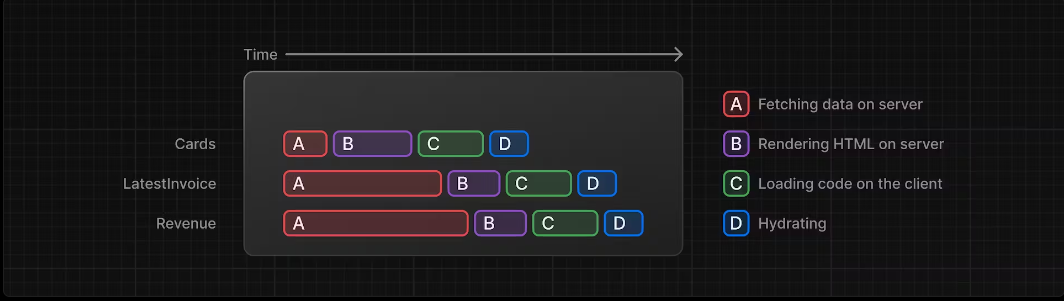
In the previous chapter, you made your dashboard page dynamic, however, we discussed how the slow data fetches can impact the performance of your application. Let's look at how you can improve the user experience when there are slow data requests.

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.



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.



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

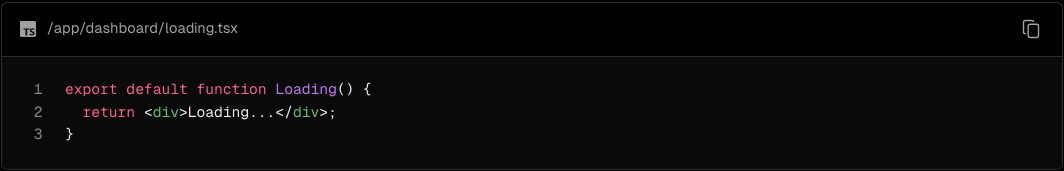
There are two ways you implement streaming in Next.js:

1. At the page level, with the loading.tsx file.
2. For specific components, with <Suspense>.

Let's see how this works.

Streaming a whole page with loading.tsx

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



Refresh http://localhost:3000/dashboard, and you should now see:



A few things are happening here:

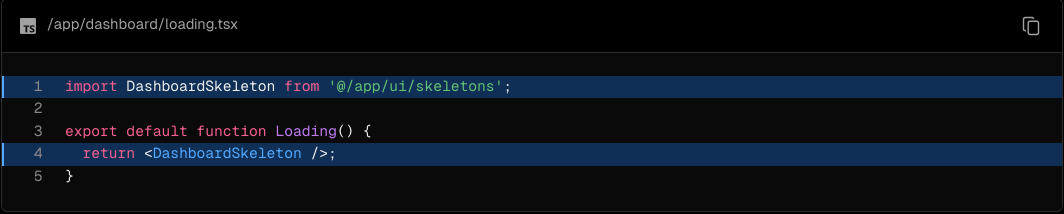
1. loading.tsx is a special Next.js file built on top of Suspense, it allows you to create fallback UI to show as a replacement while page content loads.
2. Since <Sidebar> is static, so it's shown immediately. The user can interact with <Sidebar> while the dynamic content is loading.
3. The user doesn't 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.

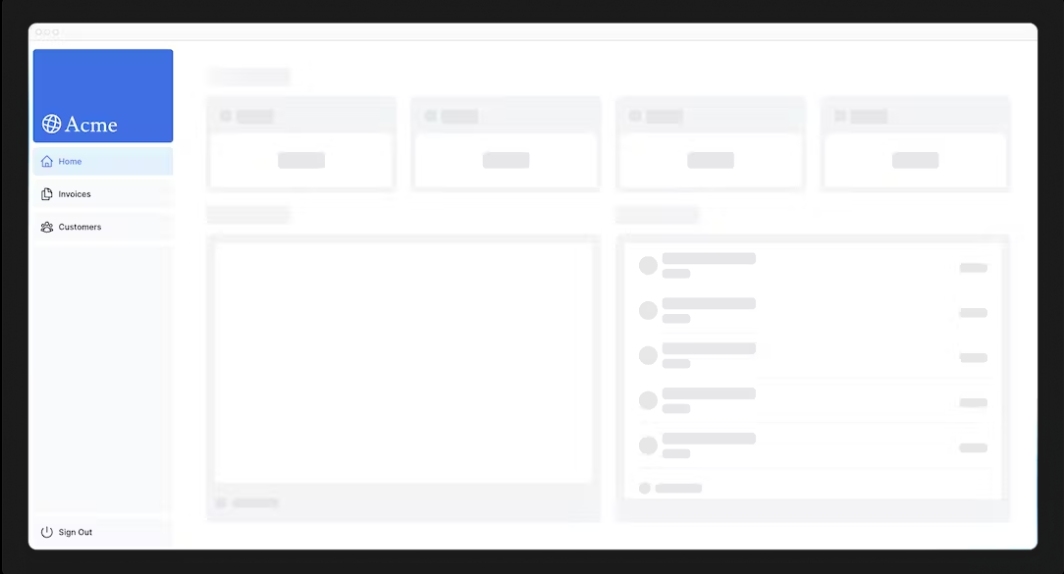
Adding loading skeletons

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>:



Then, refresh http://localhost:3000/dashboard, and you should now see:



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. Create a new folder called /(overview) inside the dashboard folder. Then, move your loading.tsx and page.tsx files inside the folder:



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

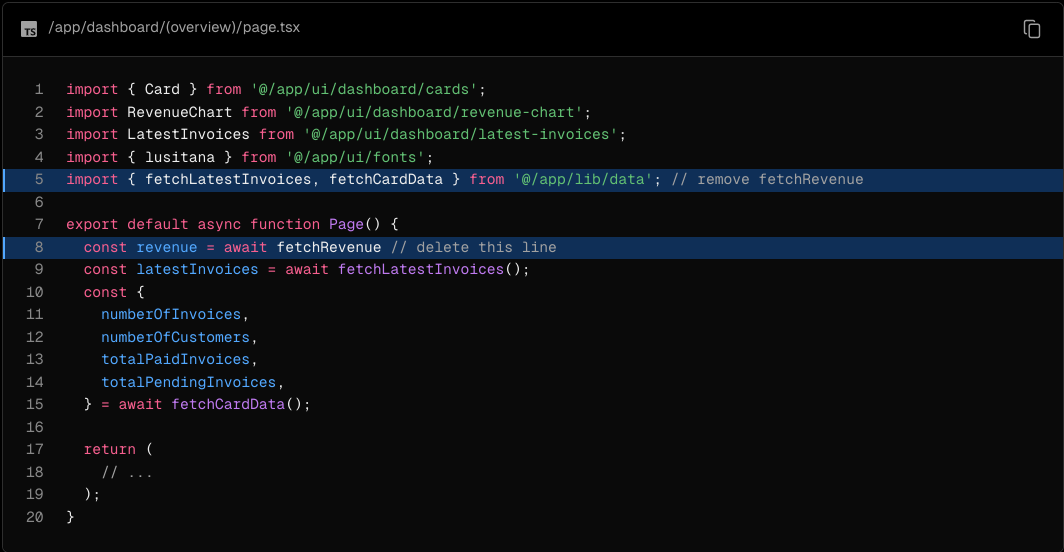
So far, you're streaming a whole page. But, instead, you can be more granular and stream specific components using React Suspense.

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.

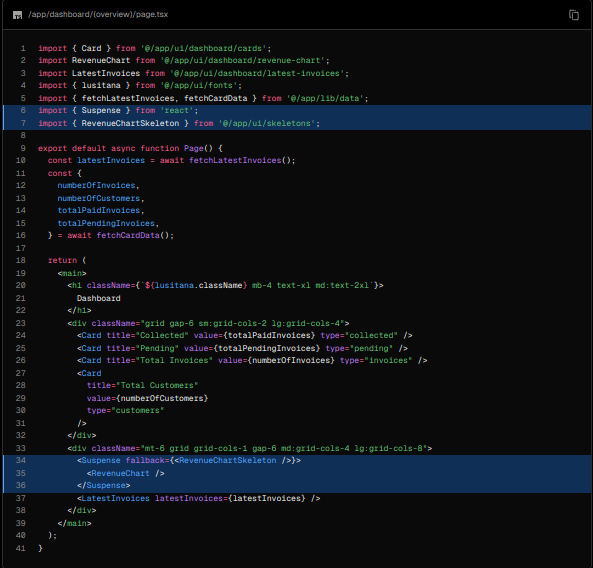
If you remember the slow data request, fetchRevenue(), this is the request that is slowing down the whole page. Instead of blocking your page, you can use Suspense to stream only this component and immediately show the rest of the page's UI.

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:

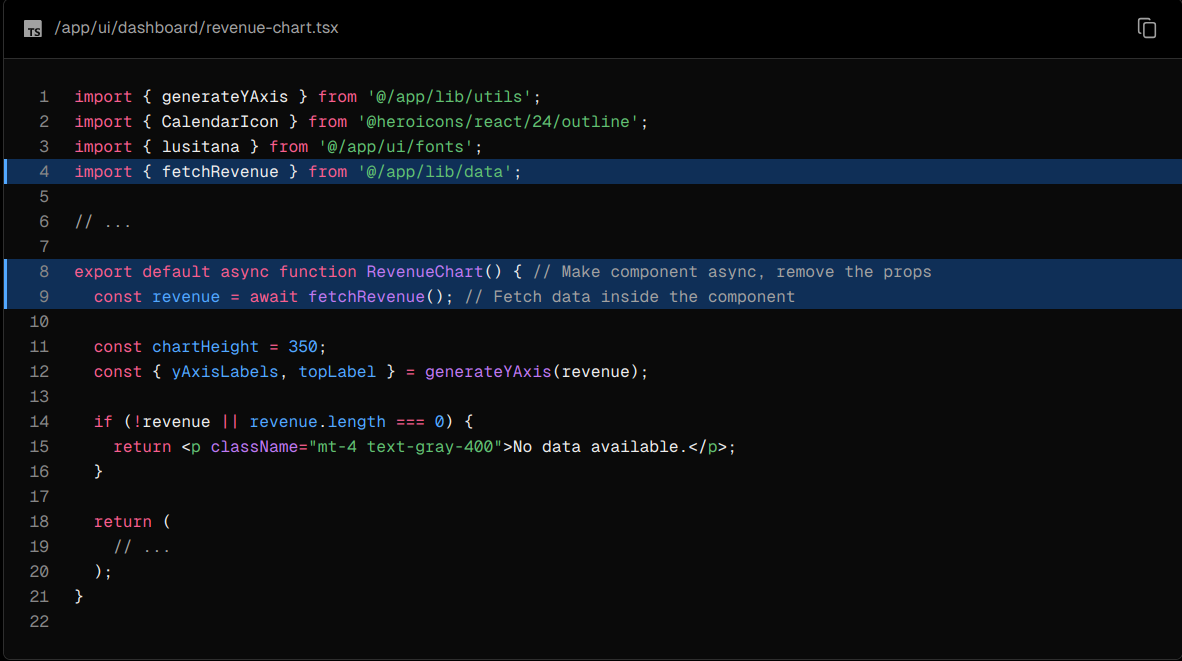
Delete all instances of fetchRevenue() and its data from /dashboard/(overview)/page.tsx:



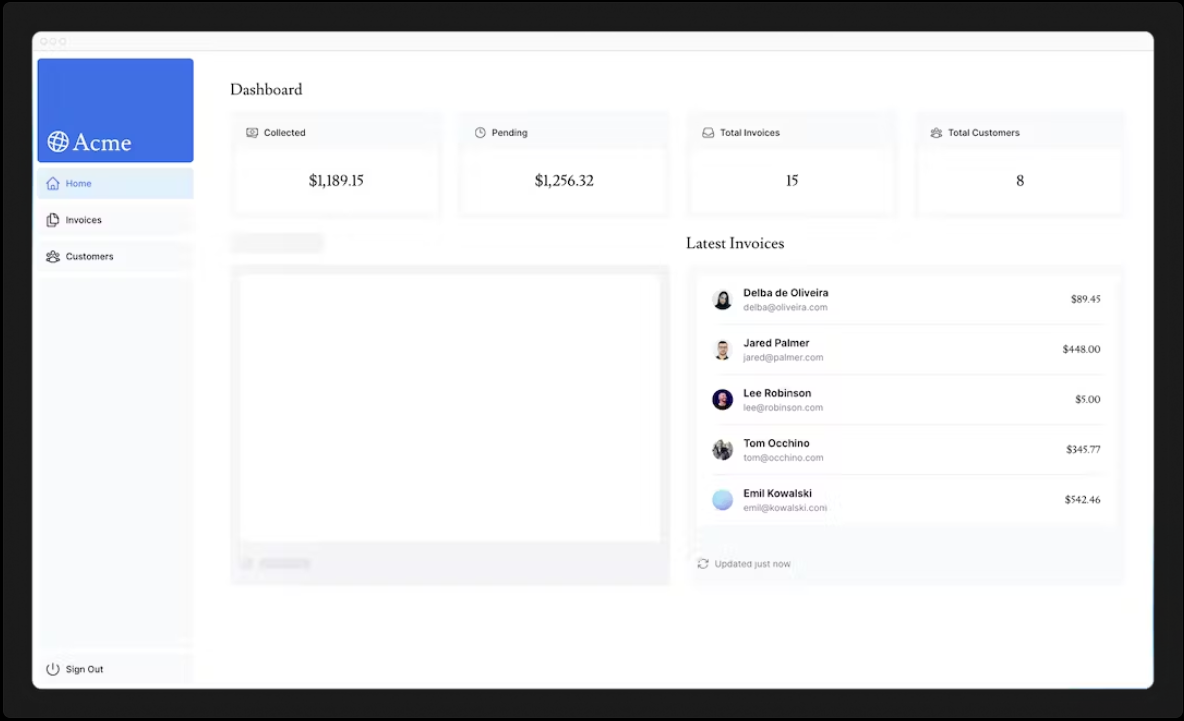
Then, import <Suspense> from React, and wrap it around <RevenueChart />. You can pass it a fallback component called <RevenueChartSkeleton>.



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



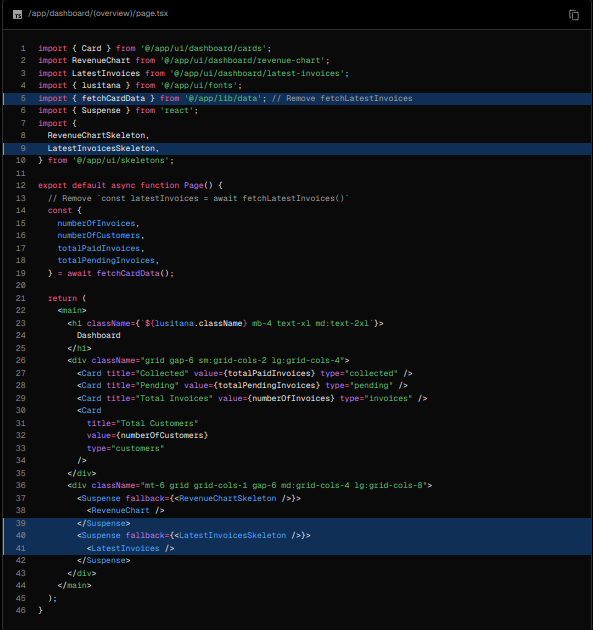
Now refresh the page, you should see the dashboard information almost immediately, while a fallback skeleton is shown for <RevenueChart>:



Practice: Streaming <LatestInvoices>

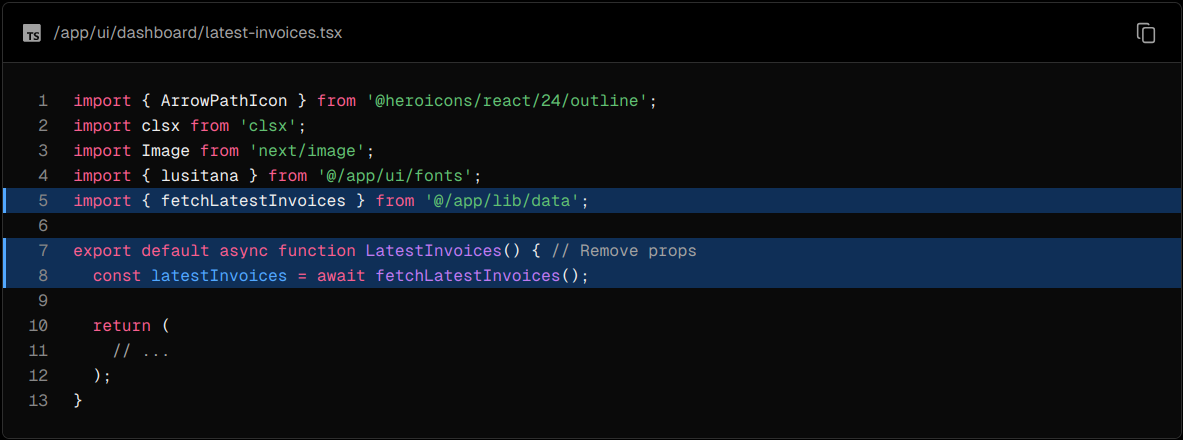
Now it's your turn! Practice what you've just learned by streaming the <LatestInvoices> component.

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:

Dashboard Page:

<LatestInvoices> component. Remember to remove the props!:



Grouping components

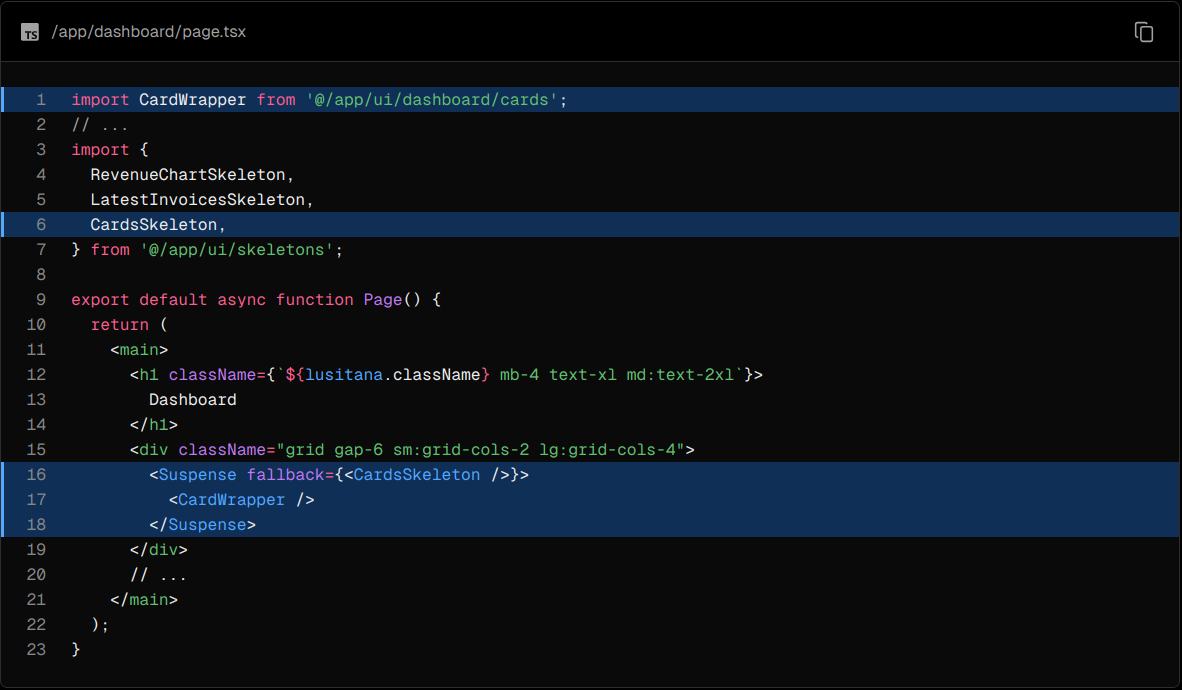
Great! You're almost there, now you need to wrap the <Card> components in Suspense. You can fetch data for each individual card, but this could lead to a popping effect as the cards load in, this can be visually jarring for the user.

So, how would you tackle this problem?

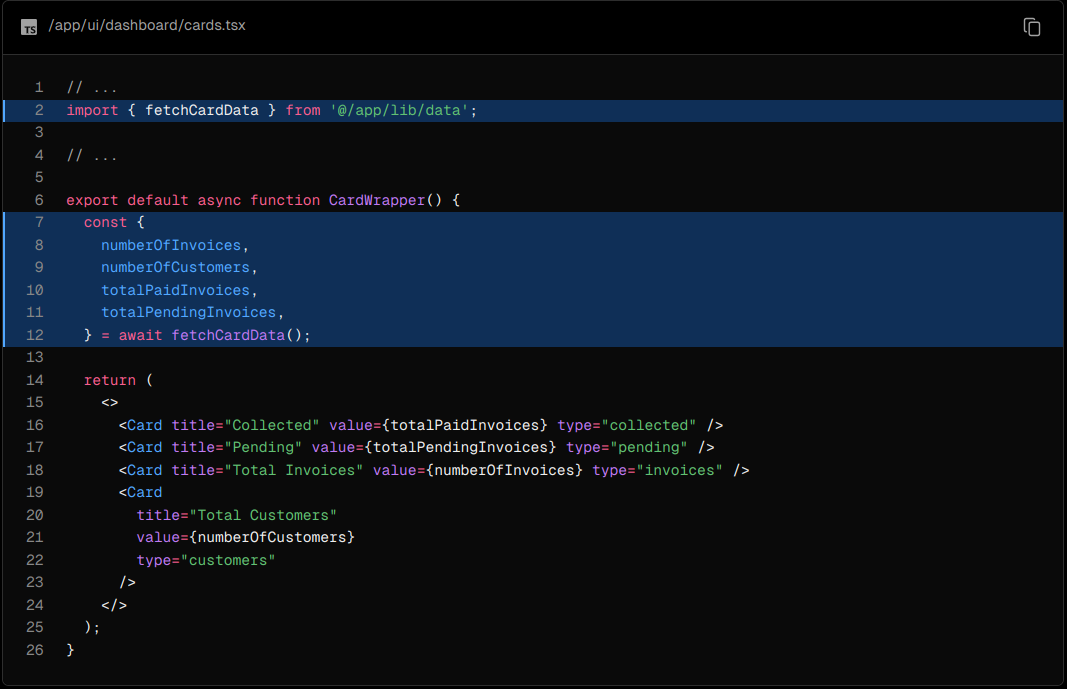
To create more of a staggered effect, you can group the cards using a wrapper component. This means the static <Sidebar/> will be shown first, followed by the cards, etc.

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.



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.



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

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.

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.

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.