Introduction to Next.js

What is Next.js?

Next.js is a React framework for building full-stack web applications. You use React Components to build user interfaces, and Next.js for additional features and optimizations.

Under the hood, Next.js also abstracts and automatically configures tooling needed for React, like bundling, compiling, and more. This allows you to focus on building your application instead of spending time with configuration.

Whether you're an individual developer or part of a larger team, Next.js can help you build interactive, dynamic, and fast React applications.

Main features of Next.js

Feature	Description
Routing	A file-system based router built on top of Server Components that supports layouts, nested routing, loading states, error handling, and more.
Rendering	Client-side and Server-side Rendering with Client and Server Components. Further optimized with Static and Dynamic Rendering on the server with Next.js. Streaming on Edge and Node.js runtimes.
Data Fetching	Simplified data fetching with async/await in Server Components, and an extended `fetch` API for request memorization, data caching and revalidation.
Styling	Support for your preferred styling methods, including CSS Modules, Tailwind CSS, and CSS-in-JS
Optimizations	Image, Fonts, and Script Optimizations to improve your application's Core Web Vitals and User Experience.
TypeScript	Improved support for TypeScript, with better type checking and more efficient compilation, as well as custom TypeScript Plugin and type checker.

History of Next.js

Next.js was created by Vercel (formerly ZEIT) to simplify the development of React applications with server-side rendering. The framework quickly gained popularity due to its performance optimizations and ease of use.

It was first released as an open-source project on GitHub on October 25, 2016

Creating first Next.js application

System requirements

- Node.js 18.18 or later.

- macOS, Windows (including WSL) and linux are supported.

Installation types

- Automatic installation
- Manual installation (we are not going to cover this in this document)

Automatic Installation

In this document we are focusing on creating Next.js applications using following command only as it will setup everything automatically for you, to create a Next.js application you need to fire following command using terminal/command prompt

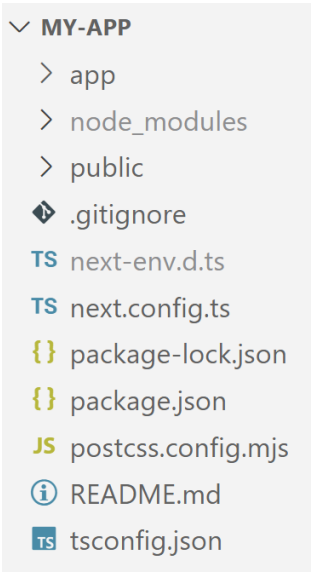
```
npx create-next-app@lateset
```

In this automatic installation process you will prompt following questions,

```
√ What is your project named? ... my-app
√ Would you like to use TypeScript? ... No / Yes
√ Would you like to use ESLint? ... No / Yes
√ Would you like to use Tailwind CSS? ... No / Yes
√ Would you like your code inside a `src/` directory? ... No / Yes
√ Would you like to use App Router? (recommended) ... No / Yes
√ Would you like to use Turbopack for `next dev`? ... No / Yes
√ Would you like to customize the import alias (`@/*` by default)? ... No / Yes
Creating a new Next.js app in D:\NextJSDemoProjects\my-app.
```

After completing the process folder with the project name will be created and all the dependencies will be installed.

Project structure



top level folders

- **app**: contains App Router
- **public**: contains static assets to be served
- **node_modules**: contains dependencies
- **src** (optional, generated if selected yes in installation process): application source folder

top level files

- **next.config.js**: configuration file for Next.js
- **tsconfig.json**: configuration fole for TypeScript
- **next-env.d.ts**: TypeScript declaration file (do not change)
- **package.json**: manifest file that describe project

Next.js Compiler

The Next.js Compiler, written in Rust using SWC (Speedy Web Compiler), allows Next.js to transform and minify your JavaScript code for production. This replaces Babel for individual files and Terser for minifying output bundles.

Compilation using the Next.js Compiler is 17x faster than Babel and enabled by default since Next.js version 12. If you have an existing Babel configuration or are using unsupported features, your application will opt-out of the Next.js Compiler and continue using Babel.

Why Next.js chosen SWC (Speedy Web Compiler)?

SWC is an extensible Rust-based platform for the next generation of fast developer tools.

SWC can be used for compilation, minification, bundling, and more – and is designed to be extended. It's something you can call to perform code transformations (either built-in or custom). Running those transformations happens through higher-level tools like Next.js.

They chose to build on SWC for a few reasons:

- **Extensibility**: SWC can be used as a Crate inside Next.js, without having to fork the library or workaround design constraints.
- **Performance**: We were able to achieve ~3x faster Fast Refresh and ~5x faster builds in Next.js by switching to SWC, with more room for optimization still in progress.
- **WebAssembly**: Rust's support for WASM is essential for supporting all possible platforms and taking Next.js development everywhere.
- **Community**: The Rust community and ecosystem are amazing and still growing.

Fast Refresh

Fast refresh is a React feature integrated into Next.js that allows you live reload the browser page while maintaining temporary client-side state when you save changes to a file. It's enabled by default in all Next.js applications on 9.4 or newer. With Fast Refresh enabled, most edits should be visible within a second.

This is how Fast Refresh works,

- If you edit a file that only exports React component(s), Fast Refresh will update the code only for that file, and re-render your component. You can edit anything in that file, including styles, rendering logic, event handlers, or effects.
- If you edit a file with exports that aren't React components, Fast Refresh will re-run both that file, and the other files importing it. So if both Button.js and Modal.js import theme.js, editing theme.js will update both components.
- Finally, if you edit a file that's imported by files outside of the React tree, Fast Refresh will fall back to doing a full reload. You might have a file which renders a React

component but also exports a value that is imported by a non-React component. For example, maybe your component also exports a constant, and a non-React utility file imports it. In that case, consider migrating the constant to a separate file and importing it into both files. This will re-enable Fast Refresh to work. Other cases can usually be solved in a similar way.

Supported Browsers

Next.js supports modern browsers with zero configuration.

- Chrome 64+
- Edge 79+
- Firefox 67+
- Opera 51+
- Safari 12+

Routing Fundamentals

Next.js uses file-system based routing, meaning you can use folders and files to define routes.

Pages

A page is UI (component) that is rendered on a specific route.

Next.js uses a file-based routing system that automatically maps files in the pages directory to application routes, supporting static, dynamic, and nested routes for seamless web development.

To create a page, add a page file inside the app directory and default export a React component. For example, to create an index page (/) we need to create page.js or page.ts or page.jsx or page.tsx file in app folder with following code in it.

app/page.tsx

```
export default function Home() {
  return (
    <h1>Hello world from Next.js</h1>
  );
}
```

To boot the server you need to fire following command from the project folder

```
npm run dev
```

This command will start web server on port number 3000 (default), and you can see the page in the browser using the url http://localhost:3000

Layout

A layout is UI that is shared between multiple pages. On navigation, layouts preserve state, remain interactive, and do not re-render.

You can define a layout by default exporting a React component from a layout file. The component should accept a children prop which can be a page or another layout.

For example, to create a layout that accepts your index page as child, add a layout file inside the app directory:

app/layout.tsx

```tsx
export default function DashboardLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>
            {/* Layout UI */}
            {/* Place children where you want to render a
                page or nested layout */}
        <main>{children}</main>
      </body>
    </html>
  )
}
```

The layout above is called a root layout because it's defined at the root of the app directory. The root layout is required and must contain html and body tags.

Creating a nested route

A nested route is a route composed of multiple URL segments.

For example, the /blog/[slug] route is composed of three segments:

- / (Root Segment)
- blog (Segment)
- [slug] (Leaf Segment)

In Next.js:

- Folders are used to define the route segments that map to URL segments.
- Files (like page and layout) are used to create UI that is shown for a segment.

To create nested routes, you can nest folders inside each other.

For example, to add a route for /blog, create a folder called blog in the app directory. Then, to make /blog publicly accessible, add a page file:

app/blog/page.tsx

```
export default function Blog() {
  return (
    <h1>Hello world from Blog Page</h1>
  );
}
```

You can continue nesting folders and create page.tsx file in each folder to create nested routes.

Creating a nested layout

By default, layouts in the folder hierarchy are also nested, which means they wrap child layouts via their children prop. You can nest layouts by adding layout inside specific route segments (folders).

For example, to create a layout for the /blog route, add a new layout file inside the blog folder.

app/blog/layout.tsx

```
export default function BlogLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return <section>{children}</section>
}
```

If you were to combine the two layouts above, the root layout (app/layout.js) would wrap the blog layout (app/blog/layout.js), which would wrap the blog (app/blog/page.js)

Linking and Navigating

There are four ways to navigate between routes in Next.js:

- Using the **<Link>** Component
- Using the **useRouter** hook (Client Components)
- Using the **redirect** function (Server Components)
- Using the native **History** API

This page will go through how to use each of these options, and dive deeper into how navigation works.

1. "<Link>" Component

<Link> is a built-in component that extends the HTML <a> tag to provide prefetching and client-side navigation between routes. It is the primary and recommended way to navigate between routes in Next.js.

You can use it by importing it from next/link, and passing a href prop to the component:

app/page.tsx

```tsx
import Link from 'next/link'

export default function Page() {
  return <Link href="/dashboard">Dashboard</Link>
}
```

The following props can be passed to the <Link> component:

Prop	Example	Type	Required
href	href="/dashboard"	String or Object	Yes
replace	replace={false}	Boolean	-
scroll	scroll={false}	Boolean	-
prefetch	prefetch={false}	Boolean or null	-

Props of Link component

href: The path or URL to navigate to.

app/page.tsx

```tsx
// Navigate to /about?name=test
export default function Page() {
  return (
    <Link
      href={{
        pathname: '/about',
        query: { name: 'test' },
      }}
    >
      About
    </Link>
  )
}
```

replace: Defaults to false. When true, next/link will replace the current history state instead of adding a new URL into the browser's history stack.

app/page.tsx

```
import Link from 'next/link'

export default function Page() {
  return <Link href="/dashboard" replace> Dashboard </Link>
}
```

scroll:

Defaults to true. The default scrolling behaviour of <Link> in Next.js is to maintain scroll position, similar to how browsers handle back and forwards navigation.

When you navigate to a new Page, scroll position will stay the same as long as the Page is visible in the viewport. However, if the Page is not visible in the viewport, Next.js will scroll to the top of the first Page element.

app/page.tsx

```
import Link from 'next/link'

export default function Page() {
  return <Link href="/dashboard" scroll={false}> Dashboard </Link>
}
```

prefetch:

Prefetching happens when a <Link /> component enters the user's viewport (initially or through scroll). Next.js prefetches and loads the linked route (denoted by the href) and its data in the background to improve the performance of client-side navigations. If the prefetched data has expired by the time the user hovers over a <Link />, Next.js will attempt to prefetch it again. Prefetching is only enabled in production.

The following values can be passed to the prefetch prop:

- null (default): Prefetch behaviour depends on whether the route is static or dynamic. For static routes, the full route will be prefetched (including all its data). For dynamic routes, the partial route down to the nearest segment with a loading.js boundary will be prefetched.
- true: The full route will be prefetched for both static and dynamic routes.
- false: Prefetching will never happen both on entering the viewport and on hover.

app/page.tsx

```tsx
import Link from 'next/link'

export default function Page() {
  return <Link href="/dashboard" prefetch={false}> Dashboard </Link>
}
```

2. "useRouter()" hook

The useRouter hook allows you to programmatically change routes from Client Components.

Recommendation: Use the <Link> component to navigate between routes unless you have a specific requirement for using useRouter.

app/page.tsx

```tsx
'use client'

import { useRouter } from 'next/navigation'

export default function Page() {
  const router = useRouter()

  return (
    <button
      type="button"
      onClick={() => router.push('/dashboard')}>
        Dashboard
    </button>
  )
}
```

3. "redirect" function

The redirect function allows you to redirect the user to another URL. redirect can be used in Server Components, Route Handlers, and Server Actions.

For Server Components, use the redirect function instead of useRouter() hook.

app/page.tsx

```tsx
import { redirect } from 'next/navigation'

export default function Home() {
  // ...
  const team = fetchTeam()
  if (!team) {
    redirect('/join')
  }

  // ...
}
```

Some important points about redirect function

- redirect returns a 307 (Temporary Redirect) status code by default.
- When used in a Server Action, it returns a 303 (See Other), which is commonly used for redirecting to a success page as a result of a POST request.
- redirect internally throws an error so it should be called outside of try/catch blocks.
- redirect can be called in Client Components during the rendering process but not in event handlers. You can use the useRouter hook instead.
- redirect also accepts absolute URLs and can be used to redirect to external links.

4. Using the native History API

Next.js allows you to use the native window.history.pushState and window.history.replaceState methods to update the browser's history stack without reloading the page.

Route Groups

In the app directory, nested folders are normally mapped to URL paths. However, you can mark a folder as a Route Group to prevent the folder from being included in the route's URL path.

This allows you to organize your route segments and project files into logical groups without affecting the URL path structure.

Route groups are useful for:

- Organizing routes into groups e.g. by site section, intent, or team.
- Enabling nested layouts in the same route segment level:
    - Creating multiple nested layouts in the same segment, including multiple root layouts

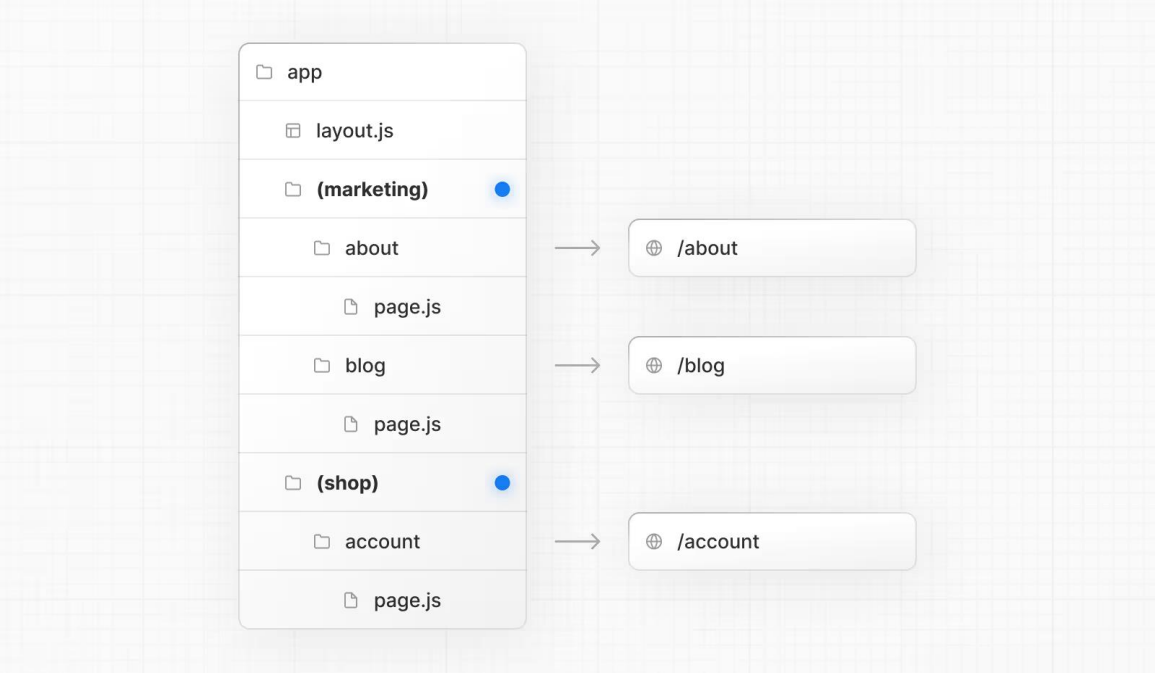
    - Adding a layout to a subset of routes in a common segment
- Adding a loading skeleton to specific route in a common segment

A route group can be created by wrapping a folder's name in parenthesis: **(folderName)**

Use cases of Route Groups:

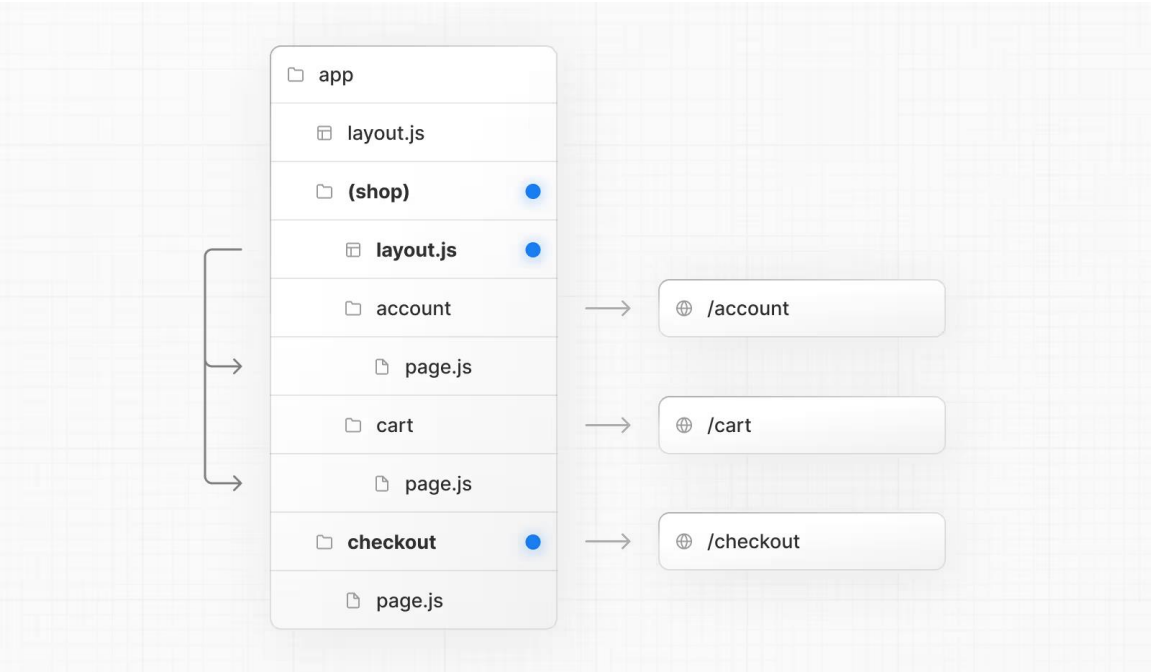
Organize routes without affecting the URL path

To organize routes without affecting the URL, create a group to keep related routes together. The folders in parenthesis will be omitted from the URL (e.g. (marketing) or (shop)).



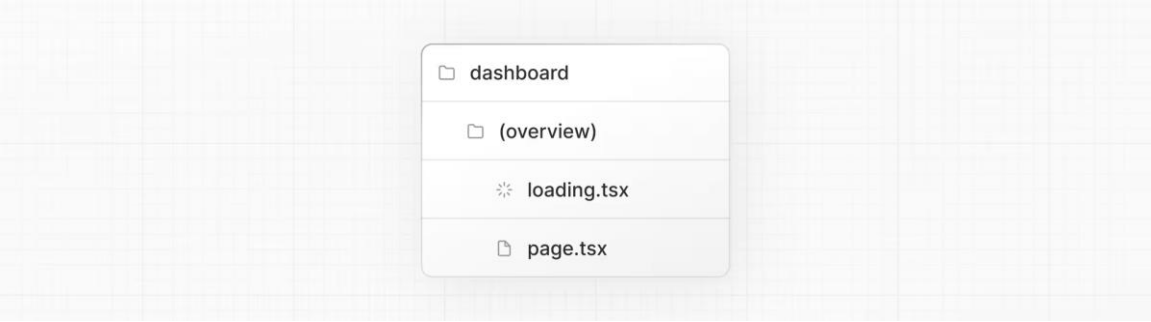
Opting specific segments into a layout

To opt specific routes into a layout, create a new route group (e.g. (shop)) and move the routes that share the same layout into the group (e.g. account and cart). The routes outside of the group will not share the layout (e.g. checkout).



Opting for loading skeletons on a specific route

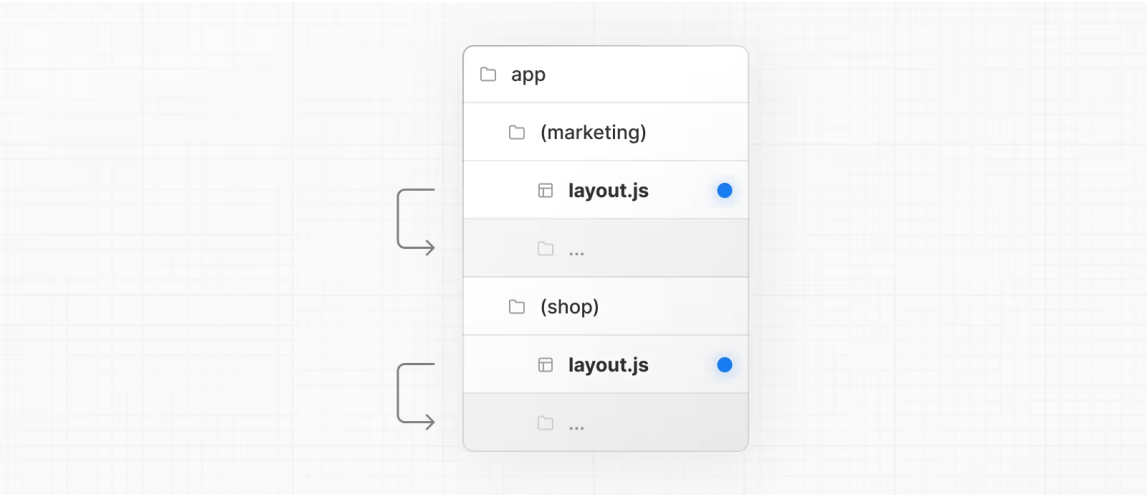
To apply a loading skeleton via a loading.js file to a specific route, create a new route group (e.g., /(overview)) and then move your loading.tsx inside that route group.



Creating multiple root layouts

To create multiple root layouts, remove the top-level layout.js file, and add a layout.js file inside each route group. This is useful for partitioning an application into sections that have a completely different UI or experience. The <html> and <body> tags need to be added to each root layout.

In the example below, both (marketing) and (shop) have their own root layout.



Important Points about Route Group

The naming of route groups has no special significance other than for organization. They do not affect the URL path.

Routes that include a route group should not resolve to the same URL path as other routes. For example, since route groups don't affect URL structure, (marketing)/about/page.js and (shop)/about/page.js would both resolve to /about and cause an error.

If you use multiple root layouts without a top-level layout.js file, your home page.js file should be defined in one of the route groups, For example: app/(marketing)/page.js.

Navigating across multiple root layouts will cause a full page load (as opposed to a client-side navigation). For example, navigating from /cart that uses app/(shop)/layout.js to /blog that uses app/(marketing)/layout.js will cause a full page load. This only applies to multiple root layouts.

Dynamic Routes (parameterized routes)

When you don't know the exact segment names ahead of time and want to create routes from dynamic data, you can use Dynamic Segments that are filled in at request time or prerendered at build time.

A Dynamic Segment can be created by wrapping a folder's name in square brackets: **[folderName]**. For example, [id] or [slug].

Dynamic Segments are passed as the params prop to layout, page, route, and generateMetadata functions.

For example, a blog could include the following route app/blog/[slug]/page.js where [slug] is the Dynamic Segment for blog posts.

app/page.tsx

```tsx
export default async function Page({
  params,
}: {
  params: Promise<{ slug: string }>
}) {
  const { slug } = await params
  return <div>My Post: {slug}</div>
}
```

Example URL	Params
/blog/arjun	{ slug : 'arjun' }
/blog/123	{ slug : '123' }

Important Points about Dynamic Routes

- Since the params prop is a promise. You must use async/await or React's use function to access the values.
- In version 14 and earlier, params was a synchronous prop. To help with backwards compatibility, you can still access it synchronously in Next.js 15, but this behaviour will be deprecated in the future.

Catch-all Segments

Dynamic Segments can be extended to catch-all subsequent segments by adding an ellipsis inside the brackets **[...folderName]**.

For example, app/shop/[...slug]/page.js will match /shop/clothes, but also /shop/clothes/tops, /shop/clothes/tops/t-shirts, and so on.

Route	Example URL	Params
app/shop/[...slug]/page.js	/shop/a	{ slug: ['a'] }
app/shop/[...slug]/page.js	/shop/a/b	{ slug: ['a', 'b'] }
app/shop/[...slug]/page.js	/shop/a/b/c	{ slug: ['a', 'b', 'c'] }

Optional Catch-all Segments

Catch-all Segments can be made optional by including the parameter in double square brackets: **[[...folderName]]**.

For example, app/shop/[[...slug]]/page.js will also match /shop, in addition to /shop/clothes, /shop/clothes/tops, /shop/clothes/tops/t-shirts.

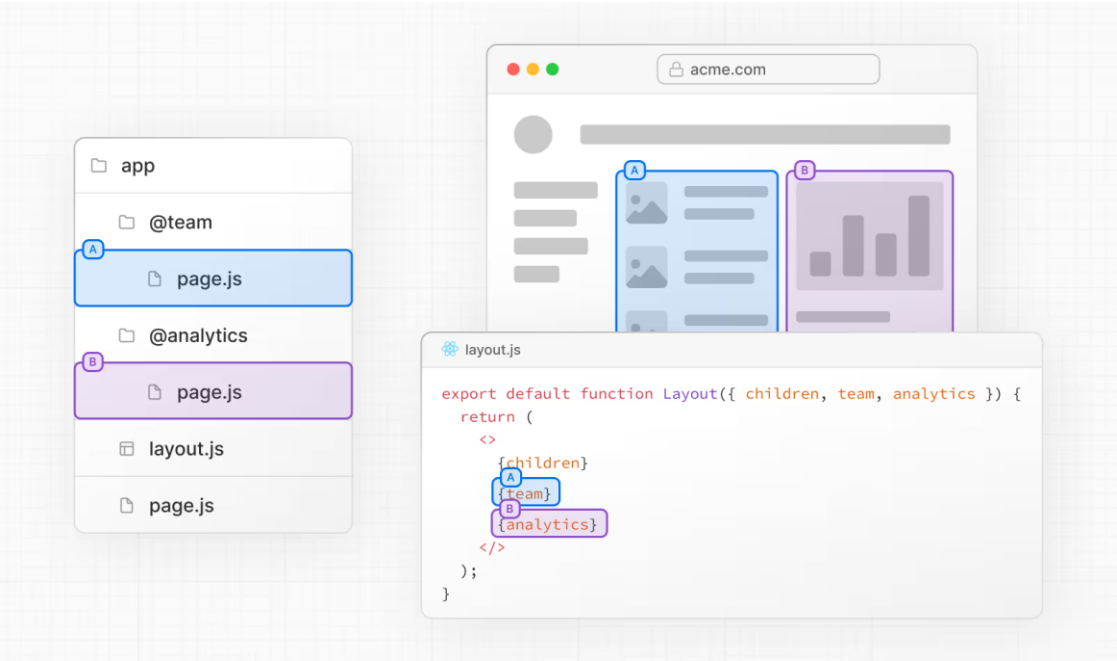
The difference between catch-all and optional catch-all segments is that with optional, the route without the parameter is also matched (/shop in the example above).

Route	Example URL	Params
app/shop/[[...slug]]/page.js	/shop	{ slug: undefined }
app/shop/[[...slug]]/page.js	/shop/a	{ slug: ['a'] }
app/shop/[[...slug]]/page.js	/shop/a/b	{ slug: ['a', 'b'] }
app/shop/[[...slug]]/page.js	/shop/a/b/c	{ slug: ['a', 'b', 'c'] }

Parallel Routes

Parallel Routes allows you to simultaneously or conditionally render one or more pages within the same layout. They are useful for highly dynamic sections of an app, such as dashboards and feeds on social sites.

For example, considering a dashboard, you can use parallel routes to simultaneously render the team and analytics pages:



Parallel routes are created using named slots. Slots are defined with the **@folder** convention. For example, the file structure above defines two slots: @analytics and @team

Slots are passed as props to the shared parent layout. For the example above, the component in app/layout.js now accepts the @analytics and @team slots props, and can render them in parallel alongside the children prop:

app/layout.tsx

```tsx
export default function Layout({
  children,
  team,
  analytics,
}: {
  children: React.ReactNode
  analytics: React.ReactNode
  team: React.ReactNode
}) {
  return (
    <>
      {children}
      {team}
      {analytics}
    </>
  )
}
```

However, slots are not route segments and do not affect the URL structure. For example, for /@analytics/views, the URL will be /views since @analytics is a slot.

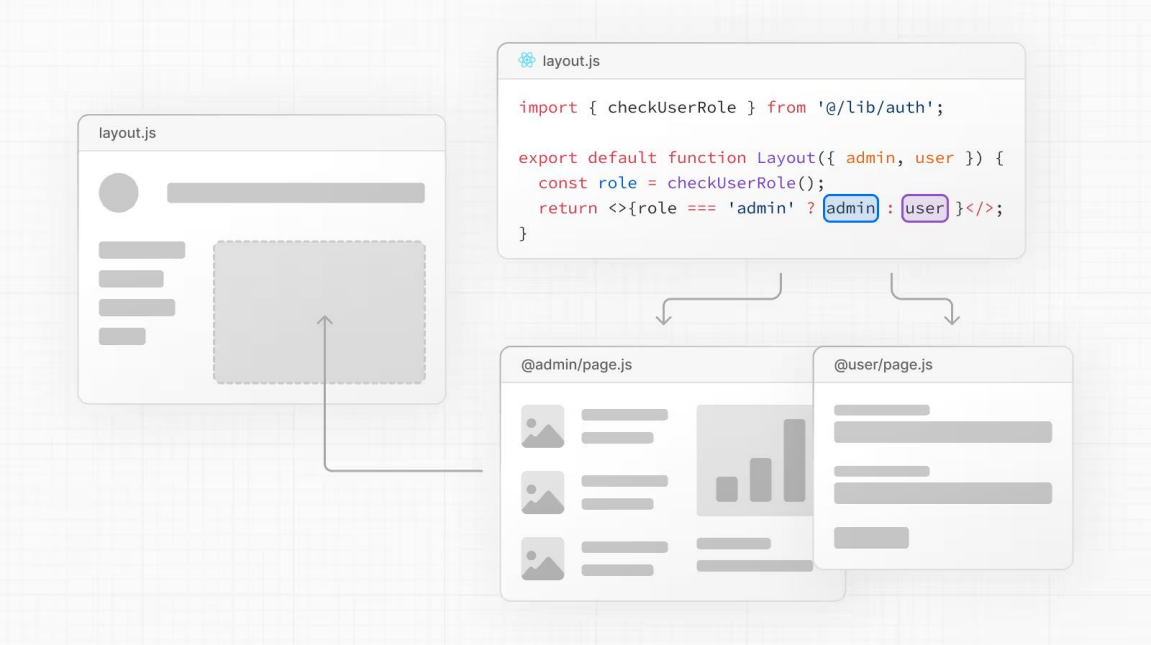
Slots are combined with the regular Page component to form the final page associated with the route segment. Because of this, you cannot have separate static and dynamic slots at the same route segment level. If one slot is dynamic, all slots at that level must be dynamic.

Use cases of parallel routes

There are many use cases of parallel routes, some of which are conditional routes, tab group, modals, loading error UI etc..

Here is an example of using parallel routes for conditional rendering.

You can use Parallel Routes to conditionally render routes based on certain conditions, such as user role. For example, to render a different dashboard page for the /admin or /user roles:



app/layout.tsx

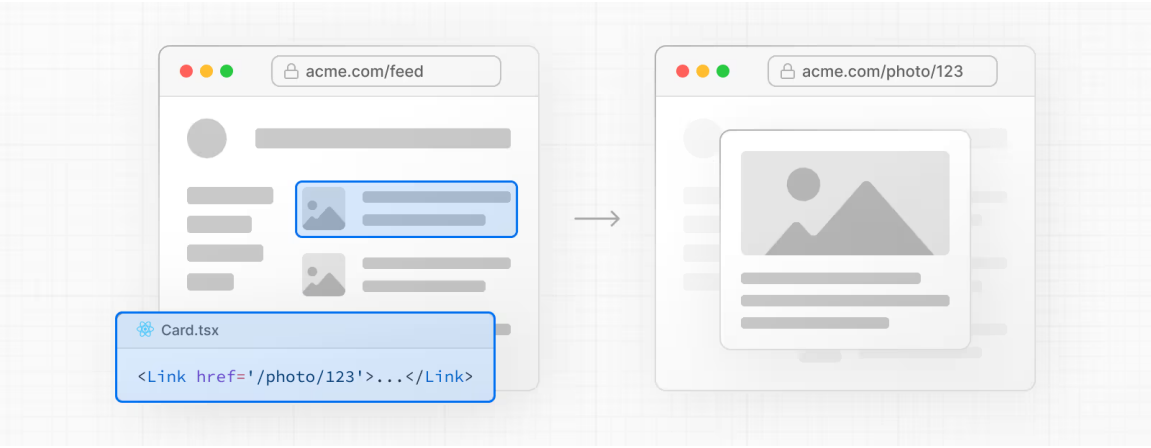
```tsx
import { checkUserRole } from '@/lib/auth'

export default function Layout({
  user,
  admin,
}: {
  user: React.ReactNode
  admin: React.ReactNode
}) {
  const role = checkUserRole()
  return role === 'admin' ? admin : user
}
```

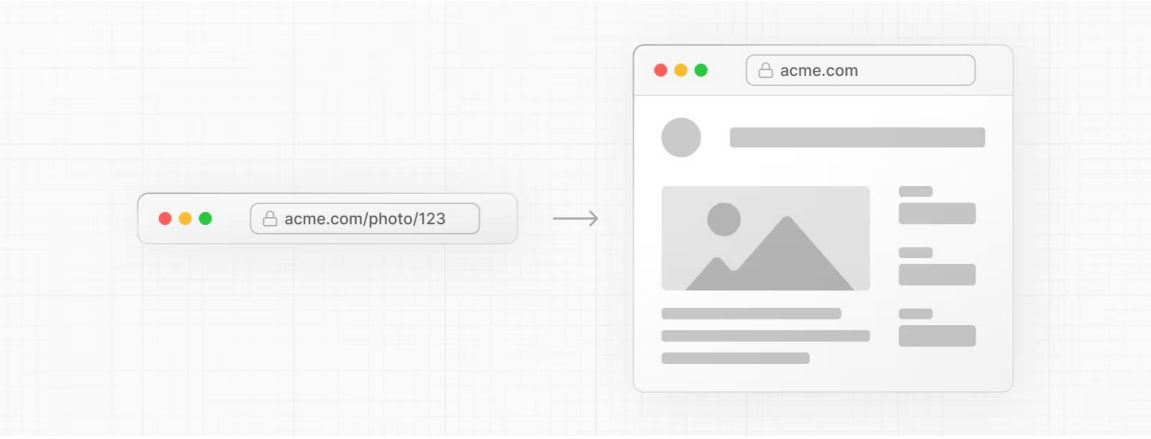
Intercepting Routes

Intercepting routes allows you to load a route from another part of your application within the current layout. This routing paradigm can be useful when you want to display the content of a route without the user switching to a different context.

For example, when clicking on a photo in a feed, you can display the photo in a modal, overlaying the feed. In this case, Next.js intercepts the /photo/123 route, masks the URL, and overlays it over /feed.



However, when navigating to the photo by clicking a shareable URL or by refreshing the page, the entire photo page should render instead of the modal. No route interception should occur.

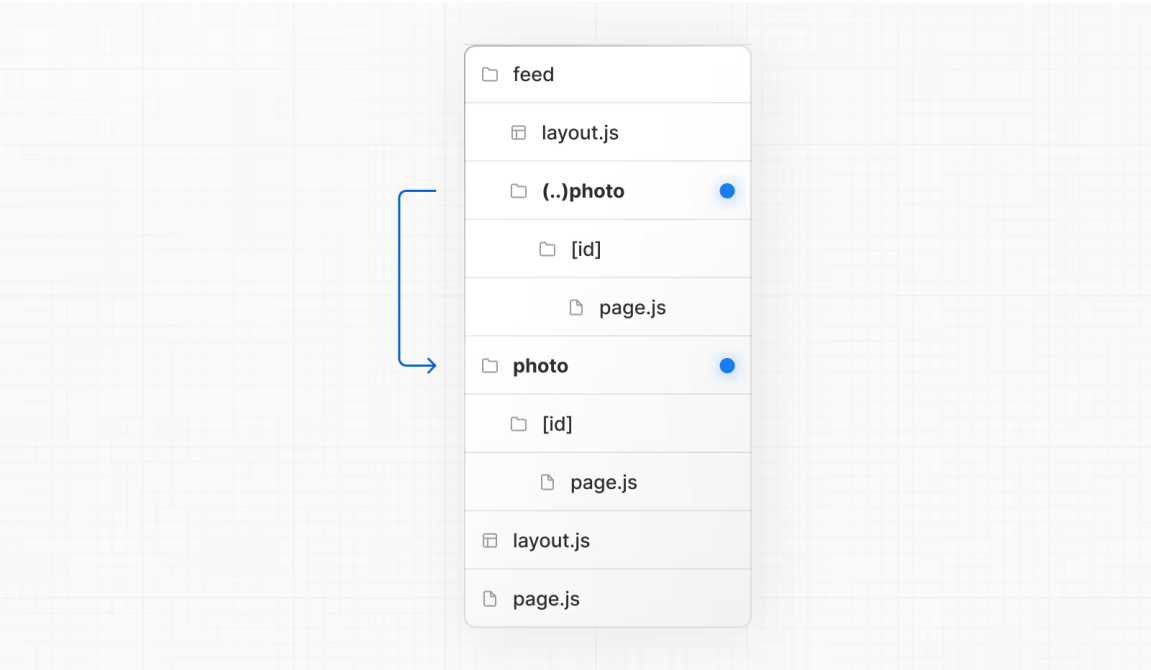


Intercepting routes can be defined with the **(..)** convention, which is similar to relative path convention ../ but for segments.

You can use:

- **(.)** to match segments on the same level
- **(..)** to match segments one level above
- **(..)(..)** to match segments two levels above
- **(...)** to match segments from the root app directory

For example, you can intercept the photo segment from within the feed segment by creating a (..)photo directory.



File Conventions

In Next.js, file naming conventions play a big role, especially when it comes to routing and structuring your application. Here's a quick guide to help you keep things clean and functional:

1. default.js

During soft navigation, Next.js keeps track of the active state (subpage) for each slot. However, for hard navigations (full-page load), Next.js cannot recover the active state. In this case, a default.js file can be rendered for subpages that don't match the current URL.

2. error.js

An error file allows you to handle unexpected runtime errors and display fallback UI.

3. layout.js

The layout file is used to define a layout in your Next.js application.

4. loading.js

A loading file can create instant loading states built on Suspense.

By default, this file is a Server Component - but can also be used as a Client Component through the "use client" directive.

app/students/loading.js

```
export default function Loading() {
    return <p>Loading...</p>
}
```

5. middleware.js

The middleware.js|ts file is used to write Middleware and run code on the server before a request is completed. Then, based on the incoming request, you can modify the response by rewriting, redirecting, modifying the request or response headers, or responding directly.

Middleware executes before routes are rendered. It's particularly useful for implementing custom server-side logic like authentication, logging, or handling redirects.

We are going to learn middlewares in great detail in later chapters.

6. not-found.js

The not-found file is used to render UI when the notFound function is thrown within a route segment. Along with serving a custom UI, Next.js will return a 200 HTTP status code for streamed responses, and 404 for non-streamed responses.

app/not-found.js

```
import Link from 'next/link'

export default function NotFound() {
  return (
    <div>
      <h2>Not Found</h2>
      <p>Could not find requested resource</p>
      <Link href="/">Return Home</Link>
    </div>
  )
}
```

7. page.js

We can create component with page.js file, we have already discussed this in great detail.

8. route.js

Route Handlers allow you to create custom request handlers for a given route using the Web Request and Response APIs.

We are going to learn Route Handlers in great detail in later chapters.

9. template.js

A template file is similar to a layout in that it wraps a layout or page. Unlike layouts that persist across routes and maintain state, templates are given a unique key, meaning children Client Components reset their state on navigation.

We have already discussed this in previous chapter.

10.unauthorized.js

The unauthorized file is used to render UI when the unauthorized function is invoked during authentication. Along with allowing you to customize the UI, Next.js will return a 401 status code.

Note: as of April 2025 this feature is experimental and subject to change, it's not recommended for production.

app/unauthorized.tsx

```tsx
import Login from '@/app/components/Login'

export default function Unauthorized() {
  return (
    <main>
      <h1>401 - Unauthorized</h1>
      <p>Please log in to access this page.</p>
      <Login />
    </main>
  )
}
```

Route Handlers

Route Handlers allow you to create custom request handlers for a given route using the Web Request and Response APIs.

Route Handlers are defined in a **route.js (or .jsx, .ts, .tsx)** file.

Route Handlers can be nested anywhere inside the app directory, similar to page.js and layout.js.

There cannot be a route.js file at the same route segment level as page.js.

app/demo/route.tsx

```tsx
export async function GET(request: Request) {
    return new Response("Created with Route Handler");
}
```

We can then access the route using /demo URL.

The following HTTP methods are supported: GET, POST, PUT, PATCH, DELETE, HEAD, and OPTIONS.

If an unsupported method is called, Next.js will return a 405 Method Not Allowed response.

In addition to supporting the native Request and Response APIs, Next.js extends them with NextRequest and NextResponse to provide convenient helpers for advanced use cases.

There are multiple use cases of Route Handlers which we will cover in later chapters.

Middlewares

Middleware allows you to run code before a request is completed. Then, based on the incoming request, you can modify the response by rewriting, redirecting, modifying the request or response headers, or responding directly.

Middleware runs before cached content and routes are matched.

Use cases of middlewares

Integrating Middleware into your application can lead to significant improvements in performance, security, and user experience.

Some common scenarios where Middleware is particularly effective include:

- Authentication and Authorization: Ensure user identity and check session cookies before granting access to specific pages or API routes.
- Server-Side Redirects: Redirect users at the server level based on certain conditions (e.g., locale, user role).
- Path Rewriting: Support A/B testing, feature rollouts, or legacy paths by dynamically rewriting paths to API routes or pages based on request properties.
- Bot Detection: Protect your resources by detecting and blocking bot traffic.
- Logging and Analytics: Capture and analyze request data for insights before processing by the page or API.
- Feature Flagging: Enable or disable features dynamically for seamless feature rollouts or testing.

Recognizing situations where middleware may **not be the optimal** approach is just as crucial. Here are some scenarios to be mindful of:

- Complex Data Fetching and Manipulation: Middleware is not designed for direct data fetching or manipulation, this should be done within Route Handlers or server-side utilities instead.
- Heavy Computational Tasks: Middleware should be lightweight and respond quickly or it can cause delays in page load. Heavy computational tasks or long-running processes should be done within dedicated Route Handlers.
- Extensive Session Management: While Middleware can manage basic session tasks, extensive session management should be managed by dedicated authentication services or within Route Handlers.

- Direct Database Operations: Performing direct database operations within Middleware is not recommended. Database interactions should be done within Route Handlers or server-side utilities.

We need to create a **middleware.ts (or .js)** file in the **root** of **your project** to define Middleware.

While **only one middleware.ts** file is supported **per project**, you can still organize your middleware logic modularly.

Break out middleware functionalities into separate .ts or .js files and import them into your main middleware.ts file.

This allows for cleaner management of route-specific middleware, aggregated in the middleware.ts for centralized control.

By enforcing a single middleware file, it simplifies configuration, prevents potential conflicts, and optimizes performance by avoiding multiple middleware layers.

middleware.ts

```
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  return NextResponse.redirect(new URL('/home', request.url))
}

export const config = {
  matcher: '/about/:path*',
}
```

Matching Paths

Middleware will be invoked for every route in your project. Given this, it's crucial to use matchers to precisely target or exclude specific routes.

The following is the execution order:

- headers from next.config.js
- redirects from next.config.js
- Middleware (rewrites, redirects, etc.)
- beforeFiles (rewrites) from next.config.js
- Filesystem routes (public/, _next/static/, pages/, app/, etc.)
- afterFiles (rewrites) from next.config.js
- Dynamic Routes (/blog/[slug])
- fallback (rewrites) from next.config.js

There are two ways to define which paths Middleware will run on:

1. Custom matcher config:
   matcher allows you to filter Middleware to run on specific paths, we can match a single path or multiple paths with an array syntax:

   middleware.ts

```
export const config = {
    matcher: ['/about/:path*', '/dashboard/:path*'],
}
```

   The matcher config allows full regex so matching like negative lookaheads or character matching is also supported.

2. Conditional statements

   middleware.ts

```
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  if (request.nextUrl.pathname.startsWith('/about')) {
    return NextResponse.rewrite(
            new URL('/about-2', request.url)
            )
  }

  if (request.nextUrl.pathname.startsWith('/dashboard')) {
    return NextResponse.rewrite(
            new URL('/dashboard/user', request.url)
            )
  }
}
```

NextResponse

The NextResponse API allows you to:

- redirect the incoming request to a different URL
- rewrite the response by displaying a given URL
- Set request headers for API Routes, getServerSideProps, and rewrite destinations
- Set response cookies
- Set response headers

To produce a response from Middleware, you can:

- rewrite to a route (Page or Route Handler) that produces a response
- return a NextResponse directly.

Rendering

Rendering converts the code you write into user interfaces.

React and Next.js allow you to create hybrid web applications where parts of your code can be rendered on the server or the client.

This section will help you understand the differences between these rendering environments, strategies, and runtimes.

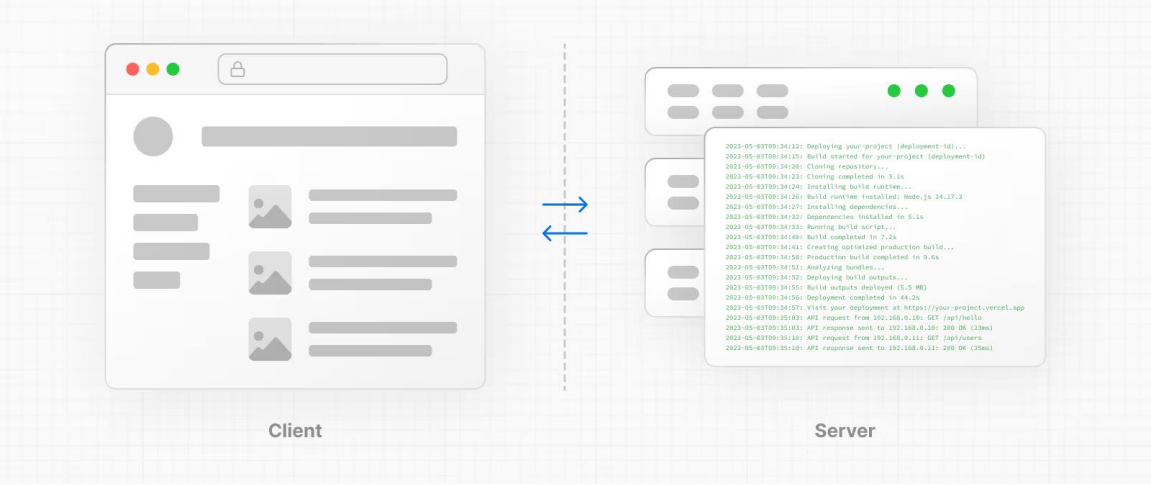
Fundamentals

To start, it's helpful to be familiar with three foundational web concepts:

- The **Environments** your application code can be executed in: the server and the client.
- The **Request-Response Lifecycle** that's initiated when a user visits or interacts with your application.
- The **Network Boundary** that separates server and client code.

Rendering Environments

There are two environments where web applications can be rendered: the client and the server.



The **client** refers to the browser on a user's device that sends a request to a server for your application code. It then turns the response from the server into a user interface.

The **server** refers to the computer in a data center that stores your application code, receives requests from a client, and sends back an appropriate response.

Historically, developers had to use different languages (e.g. JavaScript, PHP, JSP etc…) and frameworks when writing code for the server and the client.

With React, developers can use the same language (JavaScript), and the same framework (e.g. Next.js or your framework of choice). This flexibility allows you to seamlessly write code for both environments without context switching.

However, each environment has its own set of capabilities and constraints. Therefore, the code you write for the server and the client is not always the same. There are certain operations (e.g. data fetching or managing user state) that are better suited for one environment over the other.

Understanding these differences is key to effectively using React and Next.js.

Request-Response Lifecycle

Broadly speaking, all websites follow the same Request-Response Lifecycle:

1. **User Action:** The user interacts with a web application. This could be clicking a link, submitting a form, or typing a URL directly into the browser's address bar.
2. **HTTP Request:** The client sends an HTTP request to the server that contains necessary information about what resources are being requested, what method is being used (e.g. GET, POST), and additional data if necessary.
3. **Server:** The server processes the request and responds with the appropriate resources. This process may take a couple of steps like routing, fetching data, etc.
4. **HTTP Response:** After processing the request, the server sends an HTTP response back to the client. This response contains a status code (which tells the client whether the request was successful or not) and requested resources (e.g. HTML, CSS, JavaScript, static assets, etc).
5. **Client:** The client parses the resources to render the user interface.
6. **User Action:** Once the user interface is rendered, the user can interact with it, and the whole process starts again.

A major part of building a hybrid web application is deciding how to split the work in the lifecycle, and where to place the Network Boundary.

Network Boundary

In web development, the Network Boundary is a conceptual line that separates the different environments. For example, the client and the server, or the server and the data store.

In React, you choose where to place the client-server network boundary wherever it makes the most sense.

Behind the scenes, the work is split into two parts: the **client module graph** and the **server module graph**. The server module graph contains all the components that are rendered on the server, and the client module graph contains all components that are rendered on the client.

It may be helpful to think about module graphs as a visual representation of how files in your application depend on each other.

You can use the React **"use client"** convention to define the boundary. There's also a **"use server"** convention, which tells React to do some computational work on the server.

Server Components

React Server Components allow you to write UI that can be rendered and optionally cached on the server.

By default, Next.js uses Server Components. This allows you to automatically implement server rendering with no additional configuration.

app/page.tsx

```
'use server'

export default async function Home() {
  return (
    <h1>Hello world from Next.js using server component</h1>
  );
}
```

Note:

- It is optional to specify the 'use server', as it is default behaviour of next
- But, if you specify the 'use server' you have to make the function **async**

Rendering strategies

In Next.js, the rendering work is further split by route segments to enable streaming and partial rendering, and there are three different server rendering strategies:

- **Static Rendering**
  With Static Rendering, routes are rendered at build time, or in the background after data revalidation. The result is cached and can be pushed to a Content Delivery Network (CDN).
  This optimization allows you to share the result of the rendering work between users and server requests.
  Static rendering is useful when a route has data that is not personalized to the user and can be known at build time, such as a static blog post or a product page.
- **Dynamic Rendering**
  With Dynamic Rendering, routes are rendered for each user at request time.
  Dynamic rendering is useful when a route has data that is personalized to the user or has information that can only be known at request time, such as cookies or the URL's search params.

- **Streaming**
  Streaming enables you to progressively render UI from the server. Work is split into chunks and streamed to the client as it becomes ready. This allows the user to see parts of the page immediately, before the entire content has finished rendering.
  Streaming is built into the Next.js App Router by default. This helps improve both the initial page loading performance, as well as UI that depends on slower data fetches that would block rendering the whole route. For example, reviews on a product page.

Benefits of Server Rendering

There are many benefits to doing the rendering work on the server, including:

- **Data Fetching:** Server Components allow you to move data fetching to the server, closer to your data source. This can improve performance by reducing time it takes to fetch data needed for rendering, and the number of requests the client needs to make.
- **Security:** Server Components allow you to keep sensitive data and logic on the server, such as tokens and API keys, without the risk of exposing them to the client.
- **Caching:** By rendering on the server, the result can be cached and reused on subsequent requests and across users. This can improve performance and reduce cost by reducing the amount of rendering and data fetching done on each request.
- **Performance:** Server Components give you additional tools to optimize performance from the baseline. For example, if you start with an app composed of entirely Client Components, moving non-interactive pieces of your UI to Server Components can reduce the amount of client-side JavaScript needed. This is beneficial for users with slower internet or less powerful devices, as the browser has less client-side JavaScript to download, parse, and execute.
- **Initial Page Load** and **First Contentful Paint (FCP):** On the server, we can generate HTML to allow users to view the page immediately, without waiting for the client to download, parse and execute the JavaScript needed to render the page.
- **Search Engine Optimization** and **Social Network Shareability:** The rendered HTML can be used by search engine bots to index your pages and social network bots to generate social card previews for your pages.
- **Streaming:** Server Components allow you to split the rendering work into chunks and stream them to the client as they become ready. This allows the user to see parts of the page earlier without having to wait for the entire page to be rendered on the server.

How are Server Components rendered?

On the server, Next.js uses React's APIs to orchestrate rendering. The rendering work is split into chunks: by individual route segments and Suspense Boundaries.

Each chunk is rendered in two steps:

1. React renders Server Components into a special data format called the **React Server Component Payload** (RSC Payload).
2. Next.js uses the RSC Payload and Client Component JavaScript instructions to render HTML on the server.

Then, on the client:

1. The HTML is used to immediately show a fast non-interactive preview of the route - this is for the initial page load only.
2. The React Server Components Payload is used to reconcile the Client and Server Component trees, and update the DOM.
3. The JavaScript instructions are used to **hydrate** Client Components and make the application interactive.

React Server Component Payload (RSC)?

The RSC Payload is a compact binary representation of the rendered React Server Components tree. It's used by React on the client to update the browser's DOM. The RSC Payload contains:

- The rendered result of Server Components
- Placeholders for where Client Components should be rendered and references to their JavaScript files
- Any props passed from a Server Component to a Client Component

Client Components

Client Components allow you to write interactive UI that is prerendered on the server and can use client JavaScript to run in the browser.

To use Client Components, you can add the React **"use client"** directive at the top of a file, above your imports.

"use client" is used to declare a boundary between a Server and Client Component modules. This means that by defining a "use client" in a file, all other modules imported into it, including child components, are considered part of the client bundle.

app/page.tsx

```
'use client'

export default function Home() {
  return (
    <h1>Hello world from Next.js using client component </h1>
  );
}
```

Use case of client component

For example, we want to use React's useState hook which can only be used in client side rendering, if you try to use it on server component Next.js will throw an error.

app/page.tsx

```tsx
'use client'

import { useState } from 'react'

export default function Counter() {
  const [count, setCount] = useState(0)

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  )
}
```

Here, if you do not specify 'use client' directive, it will generate below error

How are Client Components Rendered?

In Next.js, Client Components are rendered differently depending on whether the request is part of a full page load (an initial visit to your application or a page reload triggered by a browser refresh) or a subsequent navigation.

Full page load:

To optimize the initial page load, Next.js will use React's APIs to render a static HTML preview on the server for both Client and Server Components.

This means, when the user first visits your application, they will see the content of the page immediately, without having to wait for the client to download, parse, and execute the Client Component JavaScript bundle.

On the server:

1. React renders Server Components into a special data format called the React Server Component Payload (RSC Payload), which includes references to Client Components.
2. Next.js uses the RSC Payload and Client Component JavaScript instructions to render HTML for the route on the server.

Then, on the client:

1. The HTML is used to immediately show a fast non-interactive initial preview of the route.
2. The React Server Components Payload is used to reconcile the Client and Server Component trees, and update the DOM.
3. The JavaScript instructions are used to **hydrate** Client Components and make their UI interactive. (Hydration is the process of attaching event listeners to the DOM, to make the static HTML interactive.)

Subsequent Navigations:

On subsequent navigations, Client Components are rendered entirely on the client, without the server-rendered HTML.

This means the Client Component JavaScript bundle is downloaded and parsed. Once the bundle is ready, React will use the RSC Payload to reconcile the Client and Server Component trees, and update the DOM.

Server and Client Composition Patterns

When building React applications, you will need to consider what parts of your application should be rendered on the server or the client.

When to use Server and Client Components?

What do you need to do?	Server Component	Client Component
Fetch data	YES	NO
Access backend resources (directly)	YES	NO

Keep sensitive information on the server (access tokens, API keys, etc)	YES	NO
Keep large dependencies on the server / Reduce client-side JavaScript	YES	NO
Add interactivity and event listeners (onClick(), onChange(), etc)	NO	YES
Use State and Lifecycle Effects (useState(), useReducer(), useEffect(), etc)	NO	YES
Use browser-only APIs	NO	YES
Use custom hooks that depend on state, effects, or browser-only APIs	NO	YES
Use React Class components	NO	YES

Runtimes

Next.js has two server runtimes you can use in your application:

- The **Node.js Runtime** (default), which has access to all Node.js APIs and compatible packages from the ecosystem.
- The **Edge Runtime** which contains a more limited set of APIs.

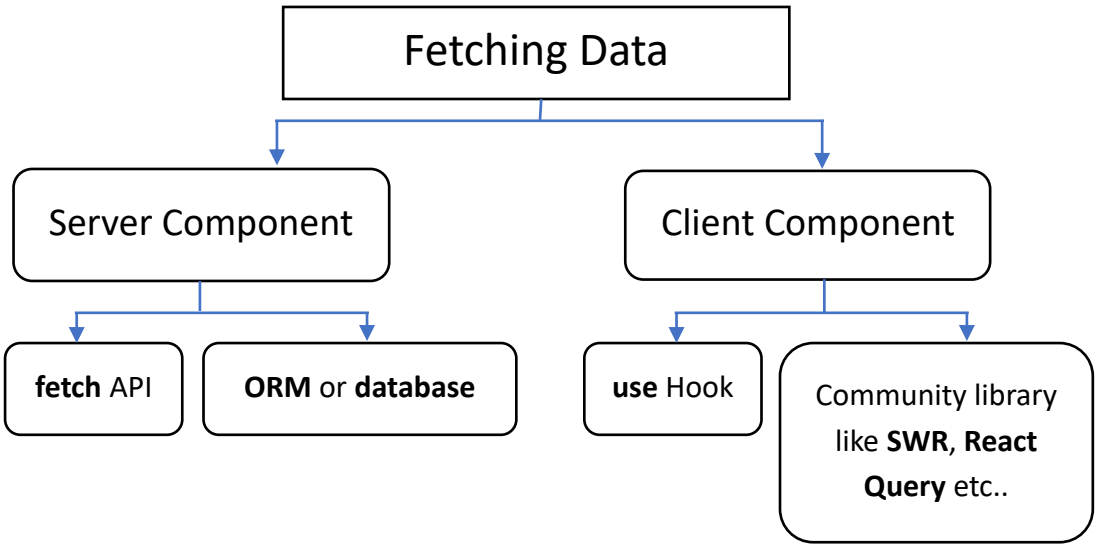
The Edge Runtime is the default runtime for Middleware. However, this can be changed to the Node.js runtime.

Fetching Data

In this section we are going to explore multiple ways by which we can fetch the data in Next.js

Basically, we can divide this topic in two parts,

1. Fetching data in server component
2. Fetching data in client component



Fetching data with fetch API

Next.js extends the Web fetch() API to allow each request on the server to set its own persistent caching and revalidation semantics.

In the browser, the cache option indicates how a fetch request will interact with the browser's HTTP cache. With this extension, cache indicates how a server-side fetch request will interact with the framework's persistent Data Cache.

You can call fetch with async and await directly within Server Components.

app/page.tsx

```tsx
export default async function Page() {
  let data = await fetch('https://api.vercel.app/blog')
  let posts = await data.json()
  return (
    <ul>
      {posts.map((post:any) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

fetch(url, options)

Since Next.js extends the Web fetch() API, you can use any of the native options available along with the new options which are explained below.

1. options.cache:

   ```
   fetch(`https://...`, { cache: 'force-cache' | 'no-store' })
   ```

   this option is used to configure how request should interact with Next.js Data Cache.

   - **auto no cache (default)**: Next.js fetches the resource from the remote server on every request in development, but will fetch once during next build because the route will be statically prerendered. If Dynamic APIs are detected on the route, Next.js will fetch the resource on every request.
   - **no-store**: Next.js fetches the resource from the remote server on every request, even if Dynamic APIs are not detected on the route.
   - **force-cache**: Next.js looks for a matching request in its Data Cache.
     - If there is a match and it is fresh, it will be returned from the cache.
     - If there is no match or a stale match, Next.js will fetch the resource from the remote server and update the cache with the downloaded resource.

2. options.next.revalidate:

   ```
   fetch(`https://...`, { next: { revalidate: false | 0 | number } })
   ```

   Set the cache lifetime of a resource (in seconds).

   - **false**: Cache the resource indefinitely. Semantically equivalent to revalidate: Infinity. The HTTP cache may evict older resources over time.
   - **0**: Prevent the resource from being cached.
   - **Number**: (in seconds) Specify the resource should have a cache lifetime of at most n seconds.

   Important points about revalidate

- If an individual fetch() request sets a revalidate number lower than the default revalidate of a route, the whole route revalidation interval will be decreased.
- If two fetch requests with the same URL in the same route have different revalidate values, the lower value will be used.
- As a convenience, it is not necessary to set the cache option if revalidate is set to a number.
- Conflicting options such as { revalidate: 3600, cache: 'no-store' } will cause an error.

3. options.next.tags:

```
fetch(`https://...`, { next: { tags: ['collection'] } })
```

Set the cache tags of a resource. Data can then be revalidated on-demand using revalidateTag. The max length for a custom tag is 256 characters and the max tag items is 128.

Fetching data from MySQL

We are going to use mysql2 package to access the data from mysql.

In order to use mysql2 we first need to install the package using following command,

```
npm install mysql2
```

After installing the package, we can directly use it in our server components, here I am just going to write all my code in the component file only but it is not recommended, ideally connection should be created in a separate file and needs to be imported and used in component.

app/page.tsx

```
import mysql from 'mysql2/promise';

export default async function Page() {
  const connection = await mysql.createConnection({
    host: 'localhost',
    user: 'root',
    database: 'testdb',
  });

  try {
    const [results, fields] = await connection.query(
      'SELECT * FROM users'
    );

    return (
      <ul>
      {results.map((user:any) => (
        <li key={user.UserID}>{user.UserName}</li>
      ))}
    </ul>
    )
  } catch (err) {
    console.log(err);
  }
}
```

Fetching data from Postgres database

We are going to use postgres package to access the data from postgres.

In order to use postgres we first need to install the package using following command,

```
npm install postgres
```

After installing the package, we can directly use it in our server components, here I am just going to write all my code in the component file only but it is not recommended, ideally connection should be created in a separate file and needs to be imported and used in component.

app/page.tsx

```tsx
import postgres from 'postgres';

interface Students{
    StudentID: number,
    StudentName: string,
    StudentRollNo: string,
    StudentDepartment: string
}
export default async function Page() {
    const conStr = 'postgres://dbuser:dbpass@localhost:5432/demodb';
    const sql = postgres(conStr);

    const students = await sql`select * from public.students`;
    return (
      <ul>
        {students.map((stu:any) => (
          <li key={stu.StudentID}>{stu.StudentName}</li>
        ))}
      </ul>
    )
}
```

Fetching data using ORM

There are many ORM (Object-Relational Mapping) available like Prisma, TypeORM etc…, we are going to use Prisma ORM, in this section we will explore how we can install and configure the Prisma for our NextJS app.

Prisma ORM is a next-generation ORM that consists of these tools:

- **Prisma Client**: Auto-generated and type-safe query builder for Node.js & TypeScript
- **Prisma Migrate**: Declarative data modeling & migration system
- **Prisma Studio**: GUI to view and edit data in your database

Prisma Client can be used in any Node.js or TypeScript backend application (including serverless applications and microservices). This can be a REST API, a GraphQL API or anything else that needs a database.

Steps to install and configure Prisma ORM:

Sr.	Step	Command
1	Install Prisma	npm install prisma
2	Initialize Prisma	npx prisma init
3	Create a database and tables in any database server like Postgres, MySQL, Oracle, sqlite, MongoDB etc…	
4	Update database connection parameters in .env file	

5	Generate Prisma schema from the database, this process is called introspect	npx prisma db pull
6	Generate Prisma client	npx prisma generate

Writing Prisma schema manually

We can also write Prisma schema manually and create a tables in our database.

The data **model** definition part of the Prisma schema defines your application models (also called Prisma models):

- Represent the entities of your application domain
- Map to the tables (relational databases like PostgreSQL) or collections (MongoDB) in your database
- Form the foundation of the queries available in the generated Prisma Client API
- When used with TypeScript, Prisma Client provides generated type definitions for your models and any variations of them to make database access entirely type safe.

We can write the Prisma Models inside the schema.prisma file.

Example of a Prisma Model:

```
model Faculty {
        id                  Int      @id @default(autoincrement())
        FacultyName         String   @db.VarChar(100)
        FacultyInitial      String   @db.VarChar(10)
}
```

After creating the Prisma Models we need to push them to the database using following command.

```
npx prisma db push
```

Prisma Studio

Prisma Studio is a visual editor for the data in your database.

Prisma Studio is included in prisma package, so we don't need to install it manually, we just need to start the studio using following command,

```
npx prisma studio
```

Fetching data using Prisma ORM

After completing the installation and configurations of Prisma we can start using the ORM in our Next.js application, for example,

app/page.tsx

```
import {PrismaClient} from '@prisma/client'
export default async function Home() {
  const prisma = new PrismaClient();
  const data = await prisma.faculty.findMany();
  return (
    <ul>
      {data.map((fac:any) => (
        <li key={fac.id}>{fac.FacultyName}</li>
      ))}
    </ul>

  );
}
```

Fetching data in client component using “use” hook

You can use React's use hook to stream data from the server to client. Start by fetching data in your Server component, and pass the promise to your Client Component as prop:

app/blog/page.tsx

```
import Posts from '@/app/ui/posts
import { Suspense } from 'react'

export default function Page() {
  // Don't await the data fetching function
  const posts = getPosts()

  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Posts posts={posts} />
    </Suspense>
  )
}
```

Then, in your Client Component, use the use hook to read the promise:

app/ui/posts.tsx

```tsx
'use client'
import { use } from 'react'

export default function Posts({
  posts,
}: {
  posts: Promise<{ id: string; title: string }[]>
}) {
  const allPosts = use(posts)

  return (
    <ul>
      {allPosts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

In the example above, you need to wrap the <Posts /> component in a <Suspense> boundary. This means the fallback will be shown while the promise is being resolved.

Fetching data in client component using community libraries

You can use a community library like SWR or React Query to fetch data in Client Components. These libraries have their own semantics for caching, streaming, and other features. For example, with SWR:

app/page.tsx

```tsx
'use client'
import useSWR from 'swr'

const fetcher = (url) => fetch(url).then((r) => r.json())

export default function BlogPage() {
  const { data, error, isLoading } = useSWR(
    'https://api.vercel.app/blog',
    fetcher
  )

  if (isLoading) return <div>Loading...</div>
  if (error) return <div>Error: {error.message}</div>

  return (
    <ul>
      {data.map((post: { id: string; title: string }) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

Server Actions and Mutations

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 include features like encrypted closures, strict input checks, error message hashing, host restrictions, and more — all working together to significantly enhance your application security.

mutations are typically used to create/update/delete data or perform server side effects.

Convention

A Server Action can be defined with the React "use server" directive. You can place the directive at the top of an async function to mark the function as a Server Action, or at the top of a separate file to mark all exports of that file as Server Actions.

Server Components can use the inline function level or module level "use server" directive. To inline a Server Action, add "use server" to the top of the function body:

app/page.tsx

```
export default function Page() {
  // Server Action
  async function create() {
    'use server'
    // Mutate data
  }

  return '...'
}
```

Alternatively, we can create Server action by creating a separate file and add the "use server" directive at the top of the file. All the exported functions will be marked as server actions that can be used in both server and client component.

app/actions.ts

```
'use server'

export async function create() {
  // code here
}
```

We can use this server actions by just importing the function like,

app/button.tsx

```tsx
'use client'

import { create } from './actions'

export function Button() {
  return <button onClick={() => create()}>Create</button>
}
```

We can also pass the server actions to a client component as a props:

app/page.tsx

```tsx
...
...
<ClientComponent updateItemAction={updateItem} />
...
...
```

app/client-component.tsx

```tsx
'use client'

export default function ClientComponent({
  updateItemAction,
}: {
  updateItemAction: (formData: FormData) => void
}) {
  return <form action={updateItemAction}>{/* ... */}</form>
}
```

Behaviour

- Server actions can be invoked using the action attribute in a <form> element:
  - Server Components support progressive enhancement by default, meaning the form will be submitted even if JavaScript hasn't loaded yet or is disabled.
  - In Client Components, forms invoking Server Actions will queue submissions if JavaScript isn't loaded yet, prioritizing client hydration.
  - After hydration, the browser does not refresh on form submission.
- Server Actions are not limited to <form> and can be invoked from event handlers, useEffect, third-party libraries, and other form elements like <button>.
- Server Actions integrate with the Next.js caching and revalidation architecture. When an action is invoked, Next.js can return both the updated UI and new data in a single server roundtrip.

- Behind the scenes, actions use the POST method, and only this HTTP method can invoke them.
- The arguments and return value of Server Actions must be serializable by React. See the React docs for a list of serializable arguments and values.
- Server Actions are functions. This means they can be reused anywhere in your application.
- Server Actions inherit the runtime from the page or layout they are used on.
- Server Actions inherit the Route Segment Config from the page or layout they are used on, including fields like maxDuration.

Form Component

The <Form> component extends the HTML <form> element to provide prefetching of loading UI, client-side navigation on submission, and progressive enhancement.

It's useful for forms that update URL search params as it reduces the boilerplate code needed to achieve the above.

Basic usage:

app/page.tsx

```tsx
import Form from 'next/form'

export default function Page() {
  return (
    <Form action="/search">
      {/* On submission, the input value will be appended to
          the URL, e.g. /search?query=abc */}
      <input name="query" />
      <button type="submit">Submit</button>
    </Form>
  )
}
```

The behaviour of the <Form> component depends on whether the action prop is passed a string or function.

When action is a string, the <Form> behaves like a native HTML form that uses a GET method. The form data is encoded into the URL as search params, and when the form is submitted, it navigates to the specified URL. In addition, Next.js:

- Prefetches the path when the form becomes visible, this preloads shared UI (e.g. layout.js and loading.js), resulting in faster navigation.

- Performs a client-side navigation instead of a full page reload when the form is submitted. This retains shared UI and client-side state.

When action is a function (Server Action), <Form> behaves like a React form, executing the action when the form is submitted.

action (string) Props

When action is a string, the <Form> component supports the following props:

Prop	Example	Type	Required
action	action="/search"	string (URL or relative path)	Yes
replace	replace={false}	boolean	No
scroll	scroll={true}	boolean	No
prefetch	prefetch={true}	boolean	No

action: The URL or path to navigate to when the form is submitted. An empty string "" will navigate to the same route with updated search params.

replace: Replaces the current history state instead of pushing a new one to the browser's history stack. Default is false.

scroll: Controls the scroll behavior during navigation. Defaults to true, this means it will scroll to the top of the new route, and maintain the scroll position for backwards and forwards navigation.

prefetch: Controls whether the path should be prefetched when the form becomes visible in the user's viewport. Defaults to true.

action (function) Props

When action is a function, the <Form> component supports the following prop:

Prop	Example	Type	Required
action	action={myAction}	function (Server Action)	Yes

action: The Server Action to be called when the form is submitted.

When action is a function, the replace and scroll props are ignored.

Example of Using forms with Server Actions to insert data

In React, you can use the action attribute in the <form> element to invoke actions. The action will automatically receive the native FormData object, containing the captured data.

For example,

app/page.tsx

```
// Server Component
export 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 has not yet loaded on the client. For example, with slower internet connections.

Server Actions are also deeply integrated with Next.js 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.

Let's see the example of creating a new student in database using NextJS.

Here are the steps you'll take to create a new student in database:

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 students page.

Let's see each step in details

1. Create a new route and form

To start, inside the /students folder, add a new route segment called /create with a page.tsx file, You'll be using this route to create new student data.

In this page we will design our form:

app/students/create/page.tsx

```
export default function Page() {
  return (
    <form>
        <input type="text" name="StudentName"/><br/>
        <input type="text" name="StudentRollNo"/><br/>
        <select name="StudentDepartment">
            <option>Computer Science and Engineering</option>
            <option>Mechanical Engineering</option>
            <option>Civil Envineering</option>
        </select><br/>
        <input type="submit"/>
    </form>
  );
}
```

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

app/students/create/page.tsx

```
export default function Page() {
    async function create(formData: FormData) {
        'use server';
        console.log("Data",formData);
    }
    return (
    <form action={create}>
        <input type="text" name="StudentName"/><br/>
        <input type="text" name="StudentRollNo"/><br/>
        <select name="StudentDepartment">
            <option>Computer Science and Engineering</option>
            <option>Mechanical Engineering</option>
            <option>Civil Envineering</option>
        </select><br/>
        <input type="submit"/>

    </form>
    );
  }
```

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

app/students/create/page.tsx

```
export default function Page() {
    async function create(formData: FormData) {
        'use server';
        const rawFormData = {
            studentName: formData.get('StudentName'),
            studentRollNo: formData.get('StudentRollNo'),
            studentDepartment: formData.get('StudentDepartment'),
        };
        console.log(rawFormData);
    }

    return (
    <form action={create}>
        <input type="text" name="StudentName"/><br/>
        <input type="text" name="StudentRollNo"/><br/>
        <select name="StudentDepartment">
            <option>Computer Science and Engineering</option>
            <option>Mechanical Engineering</option>
            <option>Civil Envineering</option>
        </select><br/>
        <input type="submit"/>

    </form>
    );
  }
```

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.

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**, a TypeScript-first validation library that can simplify this task for you.

You can download zod from npm using **npm i zod** command.

For example:

app/students/create/page.tsx

```
import { z } from 'zod';
const FormSchema = z.object({
    id: z.string(),
    studentName: z.string(),
    studentRollNo: z.coerce.number(),
    studentDepartment: z.enum(['Computer Science and Engineering',
'Mechanical Engineering', 'Civil Envineering'])
  });
const CreateStudent = FormSchema.omit({ id: true });
export default function Page() {
    async function create(formData: FormData) {
        'use server';
        const parsedData = CreateStudent.parse({
            studentName: formData.get('StudentName'),
            studentRollNo: formData.get('StudentRollNo'),
            studentDepartment: formData.get('StudentDepartment'),
        });
        console.log(parsedData);
    }

    return (
    <form action={create}>
        <input type="text" name="StudentName"/><br/>
        <input type="text" name="StudentRollNo"/><br/>
        <select name="StudentDepartment">
            <option>Computer Science and Engineering</option>
            <option>Mechanical Engineering</option>
            <option>Civil Envineering</option>
        </select><br/>
        <input type="submit"/>

    </form>
    );
  }
```

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/students/create/page.tsx

```
import { z } from 'zod';
import postgres from 'postgres';

const sql = postgres(process.env.POSTGRES_URL!);

const FormSchema = z.object({
    id: z.string(),
    studentName: z.string(),
    studentRollNo: z.coerce.number(),
    studentDepartment: z.enum(['Computer Science and Engineering',
'Mechanical Engineering', 'Civil Envineering'])
  });
const CreateStudent = FormSchema.omit({ id: true });
export default function Page() {
    async function create(formData: FormData) {
        'use server';
        const {studentName, studentRollNo, studentDepartment} =
CreateStudent.parse({
            studentName: formData.get('StudentName'),
            studentRollNo: formData.get('StudentRollNo'),
            studentDepartment: formData.get('StudentDepartment'),
        });
        try{
            await sql`INSERT INTO public.students ("studentName",
"studentRollNo", "studentDepartment") VALUES (${studentName},
${studentRollNo}, ${studentDepartment})`;
        }catch(e){
            console.log(e)
        }
    }

    return (
    <form action={create}>
        <input type="text" name="StudentName"/><br/>
        <input type="text" name="StudentRollNo"/><br/>
        <select name="StudentDepartment">
            <option>Computer Science and Engineering</option>
            <option>Mechanical Engineering</option>
            <option>Civil Envineering</option>
        </select><br/>
        <input type="submit"/>

    </form>
    );
  }
```

6. Revalidate and redirect

Next.js has a client-side router cache that stores the route segments in the user's browser for a time. Along with 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 students route, you want to clear this cache and trigger a new request to the server. You can do this with the revalidatePath function from Next.js.

Once the database has been updated, the /students 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 /students page. You can do this with the redirect function from Next.js:

app/students/create/page.tsx

```
import { revalidatePath } from 'next/cache';
import { redirect } from 'next/navigation';
...
...
export default function Page() {
    async function create(formData: FormData) {
        'use server';
         ...
         ...
        revalidatePath('/students');
        redirect('/students');
    }

    return (
    <form action={create}>
       ...
       ...
    </form>
    );
  }
```

Note: we can not write redirect from the try-catch block as redirect works by throwing an error so when you write it in the catch block, you're just catching it and it won't work as expected.

Example of Using forms with Server Actions to update data

The updating invoice form is similar to the create a student form, except you'll need to pass the student id to update the record in your database. Let's see how you can get and pass the student id.

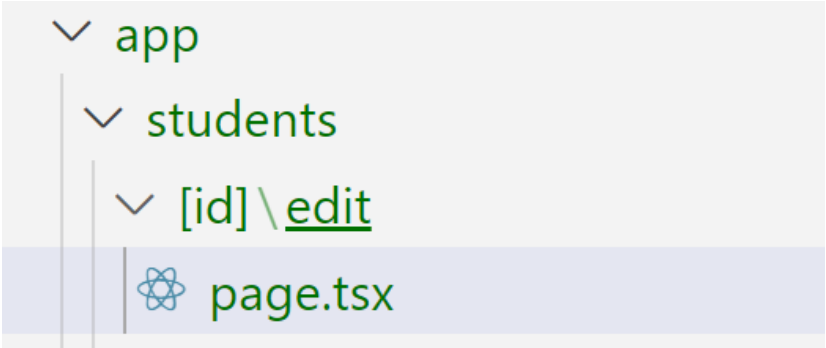
These are the steps you'll take to update an invoice:

1. Create a new dynamic route segment with the student id.
2. Read the student id from the page params.
3. Fetch the specific student from your database.
4. Pre-populate the form with the student data.
5. Update the student data in your database.

1. Create a new dynamic route segment with the student id.

Next.js allows you to create Dynamic Route Segments 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 /students 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:



2. Read the student id from the page params.

app/students/[id]/edit/page.tsx

```tsx
async function Page(props: { params: Promise<{ id: string }> }){
    const {id} = await props.params;
    console.log("ID = ",id);
}
export default Page;
```

3. Fetch the specific student from database

app/students/[id]/edit/page.tsx

```
import postgres from 'postgres';
const sql = postgres(process.env.POSTGRES_URL!);
async function Page(props: { params: Promise<{ id: string }> }){
    const {id} = await props.params;
    const data = await sql`select * from public.students where
id=${params.id}`;
    }
}
export default Page;
```

4. Pre-populate the form with the student data.

app/students/[id]/edit/page.tsx

```
import postgres from 'postgres';
const sql = postgres(process.env.POSTGRES_URL!);

async function Page(props: { params: Promise<{ id: string }> }){
    const {id} = await props.params;
    console.log("ID = ",id);
    const data = await sql`select * from public.students where
id=${id}`;
    return(
        <>
            <h1>Edit Page here</h1>
            <form>
                <input type="text" name="StudentName"
defaultValue={data[0].studentName}/><br/>
                <input type="text" name="StudentRollNo"
defaultValue={data[0].studentRollNo}/><br/>
                <select name="StudentDepartment"
defaultValue={data[0].studentDepartment}>
                    <option>Computer Science and Engineering</option>
                    <option>Mechanical Engineering</option>
                    <option>Civil Envineering</option>
                </select><br/>
                <input type="submit"/>

            </form>
        </>
    )
}
export default Page;
```

5. Update the student data in your database.

app/students/[id]/edit/page.tsx

```
// imports like insert here
const sql = postgres(process.env.POSTGRES_URL!);
// validation like insert here

async function Page(props: { params: Promise<{ id: string }> }){
    const {id} = await props.params;
    const data = await sql`select * from public.students where
id=${id}`;
    async function update(formData: FormData) {
        'use server';

        const {studentName, studentRollNo, studentDepartment} =
CreateStudent.parse({
            studentName: formData.get('StudentName'),
            studentRollNo: formData.get('StudentRollNo'),
            studentDepartment: formData.get('StudentDepartment')
        });
        try{
            await sql`UPDATE public.students SET
"studentName"=${studentName}, "studentRollNo"=${studentRollNo},
"studentDepartment"=${studentDepartment} where id=${id}`;
        }catch(e){
            console.log(e)
        }
        revalidatePath('/students');
        redirect('/students');
    }

    return(
        <>
            <form action={update}>
                ...
            </form>
        </>
    )
}
export default Page;
```

Deleting data in client component using community libraries

When we need use interactions in our component like handling key/mouse events we need to create a client component and call the server action from it.

To perform delete operation on our student example we need to create client component called DeleteButton, define a button there and call a server action from it created at other file named DeleteStudent.tsx, here is the files and code required for delete operation.

app/lib/DeleteStudent.tsx

```
"use server"
import { revalidatePath } from 'next/cache';
import postgres from 'postgres';

const sql = postgres(process.env.POSTGRES_URL!);
export async function deleteStu(props:any){
    try{
        await sql`DELETE FROM public.students where id=${props.id}`;
    }catch(e){
        console.log(e)
    }
    revalidatePath('/students');
}
```

app/ui/students/buttons.tsx

```
"use client"
import {deleteStu} from '../../lib/DeleteStudent'
export function DeleteButton(id:any){
    return(
        <button onClick={async ()=>deleteStu(id)}>Delete</button>
    );
}
```

app/students/page.tsx

```
...
import { DeleteButton } from '../ui/students/buttons';
...
...
    <tr key={index}>
        <td>{stu.studentName}</td>
        <td>{stu.studentRollNo}</td>
        <td>{stu.studentDepartment}</td>
        <td><Link href={`/students/${stu.id}/edit`}>Edit</Link></td>
        <td><DeleteButton key={"deleteBtn"+index} id={stu.id} /></td>
    </tr>
...
...
```

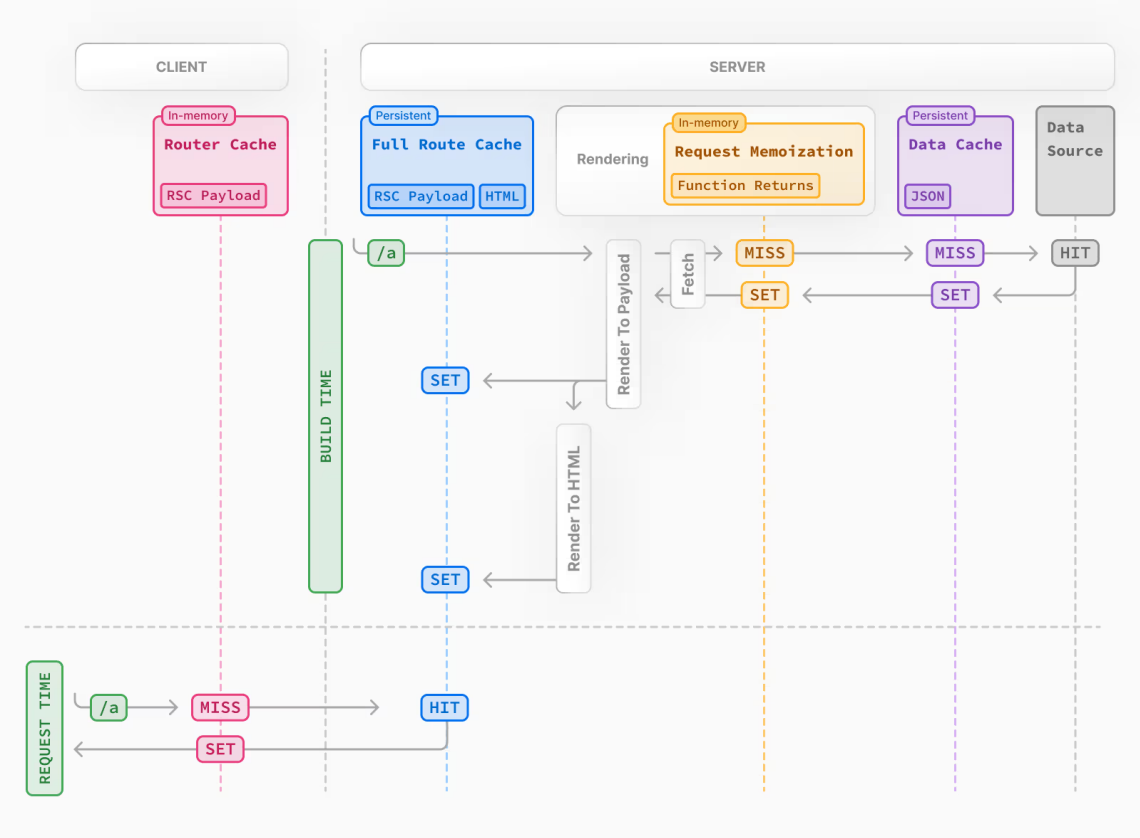
Caching

Next.js improves your application's performance and reduces costs by caching rendering work and data requests. This chapter provides an in-depth look at Next.js caching mechanisms, the APIs you can use to configure them, and how they interact with each other.

Here's a high-level overview of the different caching mechanisms and their purpose:

Mechanism	What	Where	Purpose	Duration
Request Memoization	Return values of functions	Server	Re-use data in a React Component tree	Per-request lifecycle
Data Cache	Data	Server	Store data across user requests and deployments	Persistent (can be revalidated)
Full Route Cache	HTML and RSC payload	Server	Reduce rendering cost and improve performance	Persistent (can be revalidated)
Router Cache	RSC Payload	Client	Reduce server requests on navigation	User session or time-based

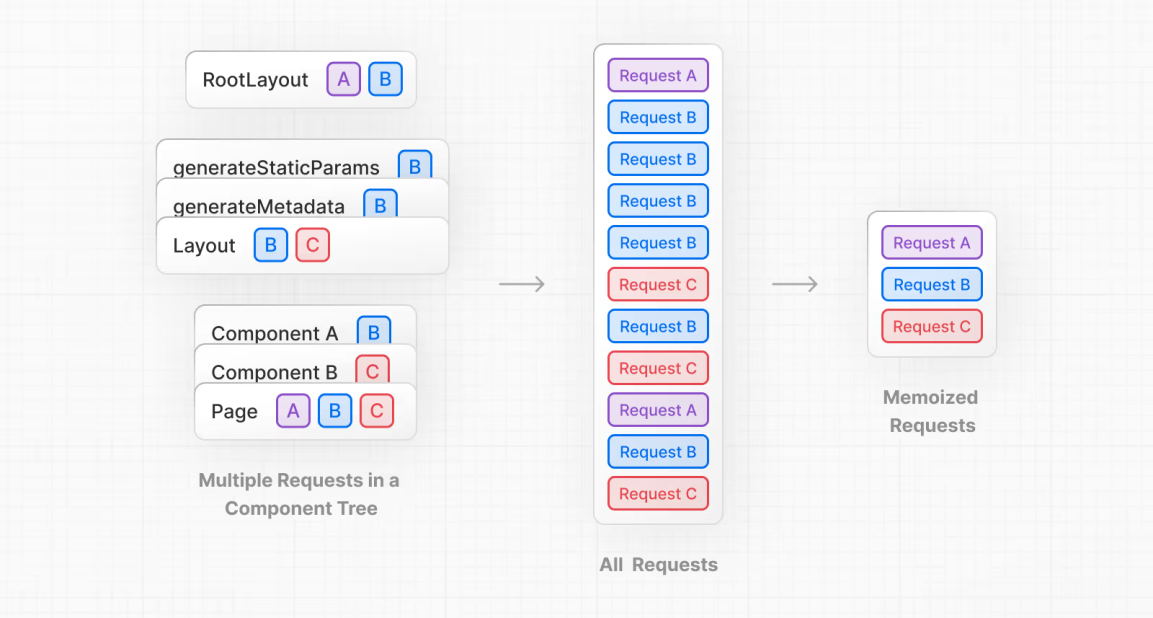
By default, Next.js will cache as much as possible to improve performance and reduce cost. This means routes are statically rendered and data requests are cached unless you opt out. The diagram below shows the default caching behaviour: when a route is statically rendered at build time and when a static route is first visited.



Caching behaviour changes depending on whether the route is statically or dynamically rendered, data is cached or uncached, and whether a request is part of an initial visit or a subsequent navigation. Depending on your use case, you can configure the caching behaviour for individual routes and data requests.

Request Memoization

Next.js extends the fetch API to automatically memoize requests that have the same URL and options. This means you can call a fetch function for the same data in multiple places in a React component tree while only executing it once.



For example, if you need to use the same data across a route (e.g. in a Layout, Page, and multiple components), you do not have to fetch data at the top of the tree, and forward props between components. Instead, you can fetch data in the components that need it without worrying about the performance implications of making multiple requests across the network for the same data.

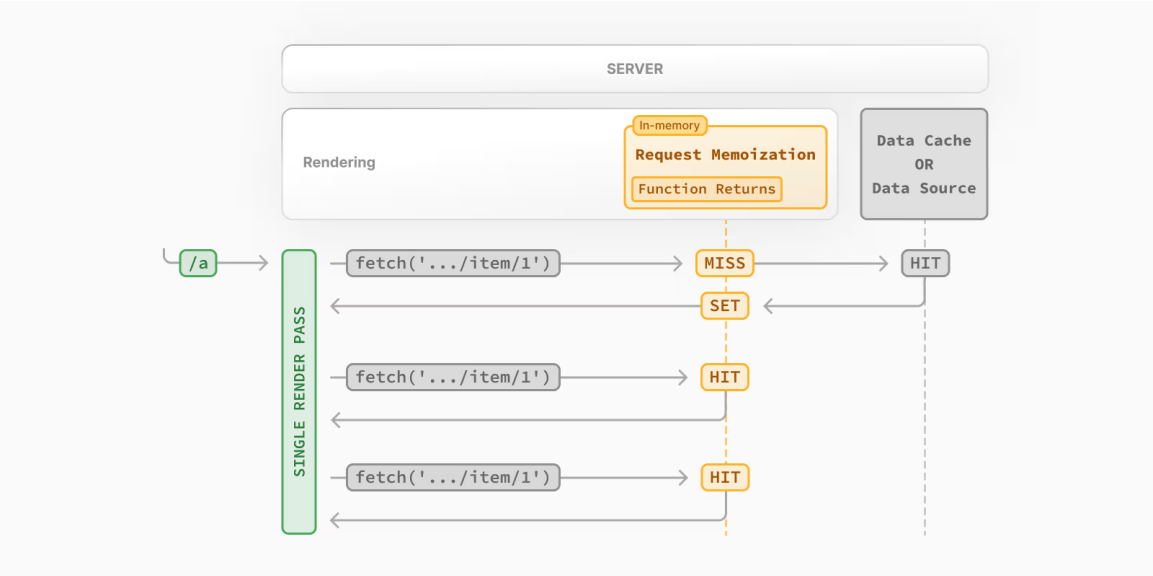
app/students/page.tsx

```tsx
async function getItem() {
    // The `fetch` function is automatically memoized and the result
    // is cached
    const res = await fetch('https://.../item/1')
    return res.json()
}

// This function is called twice, but only executed the first time
const item = await getItem() // cache MISS

// The second call could be anywhere in your route
const item = await getItem() // cache HIT
```

How Request Memoization Works:



- While rendering a route, the first time a particular request is called, its result will not be in memory and it'll be a cache MISS.
- Therefore, the function will be executed, and the data will be fetched from the external source, and the result will be stored in memory.
- Subsequent function calls of the request in the same render pass will be a cache HIT, and the data will be returned from memory without executing the function.
- Once the route has been rendered and the rendering pass is complete, memory is "reset" and all request memoization entries are cleared.
- Request memoization is a React feature, not a Next.js feature. It's included here to show how it interacts with the other caching mechanisms.
- Memoization only applies to the GET method in fetch requests.
- Memoization only applies to the React Component tree, this means:

- It applies to fetch requests in generateMetadata, generateStaticParams, Layouts, Pages, and other Server Components.
- It doesn't apply to fetch requests in Route Handlers as they are not a part of the React component tree.

Duration

The cache lasts the lifetime of a server request until the React component tree has finished rendering.

Revalidating

Since the memoization is not shared across server requests and only applies during rendering, there is no need to revalidate it.

Opting Out

Memoization only applies to the GET method in fetch requests, other methods, such as POST and DELETE, are not memoized. This default behavior is a React optimization and we do not recommend opting out of it.

To manage individual requests, you can use the signal property from AbortController. However, this will not opt requests out of memoization, rather, abort in-flight requests.

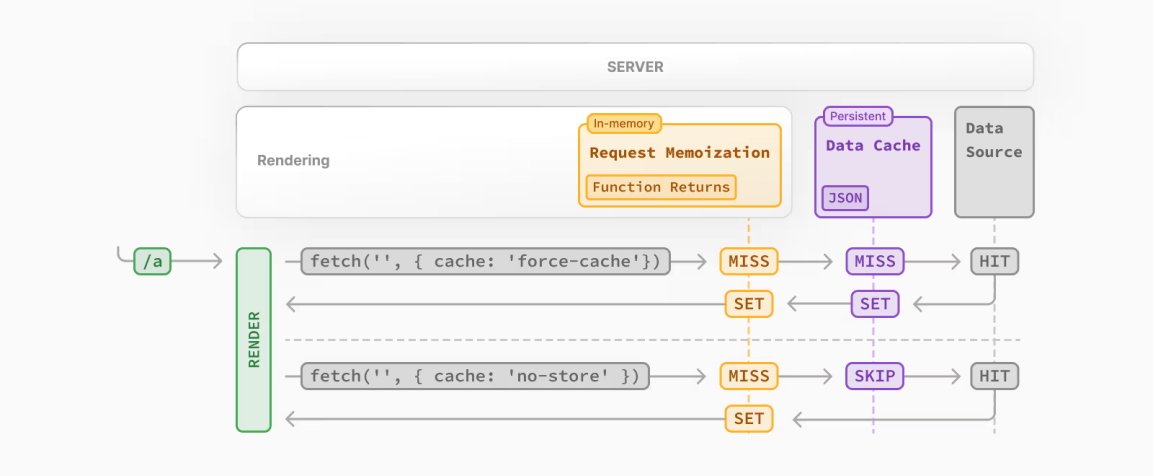
```
const { signal } = new AbortController()
fetch(url, { signal })
```

Data Cache

Next.js has a built-in Data Cache that persists the result of data fetches across incoming server requests and deployments. This is possible because Next.js extends the native fetch API to allow each request on the server to set its own persistent caching semantics.

You can use the cache and next.revalidate options of fetch to configure the caching behaviour.

How the Data Cache Works



The first time a fetch request with the 'force-cache' option is called during rendering, Next.js checks the Data Cache for a cached response.

If a cached response is found, it's returned immediately and memoized.

If a cached response is not found, the request is made to the data source, the result is stored in the Data Cache, and memoized.

For uncached data (e.g. no cache option defined or using { cache: 'no-store' }), the result is always fetched from the data source, and memoized.

Whether the data is cached or uncached, the requests are always memoized to avoid making duplicate requests for the same data during a React render pass.

Duration

The Data Cache is persistent across incoming requests and deployments unless you revalidate or opt-out.

Revalidating

Cached data can be revalidated in two ways, with:

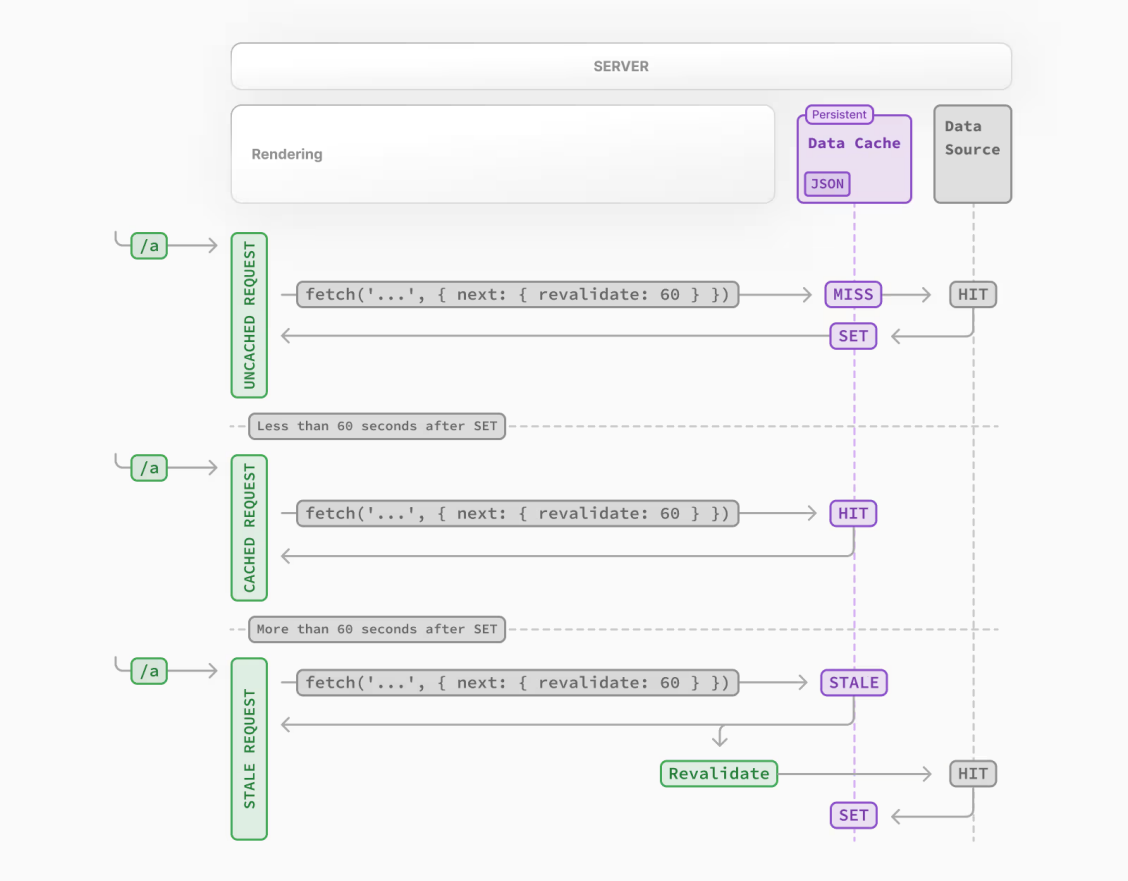
- **Time-based Revalidation:** Revalidate data after a certain amount of time has passed and a new request is made. This is useful for data that changes infrequently and freshness is not as critical.
- **On-demand Revalidation:** Revalidate data based on an event (e.g. form submission). On-demand revalidation can use a tag-based or path-based approach to revalidate groups of data at once. This is useful when you want to ensure the latest data is shown as soon as possible (e.g. when content from your headless CMS is updated).

Time-based Revalidation:

To revalidate data at a timed interval, you can use the next.revalidate option of fetch to set the cache lifetime of a resource (in seconds).

```
// Revalidate at most every hour
fetch('https://...', { next: { revalidate: 3600 } })
```

Alternatively, you can use Route Segment Config options to configure all fetch requests in a segment or for cases where you're not able to use fetch.



The first time a fetch request with revalidate is called, the data will be fetched from the external data source and stored in the Data Cache.

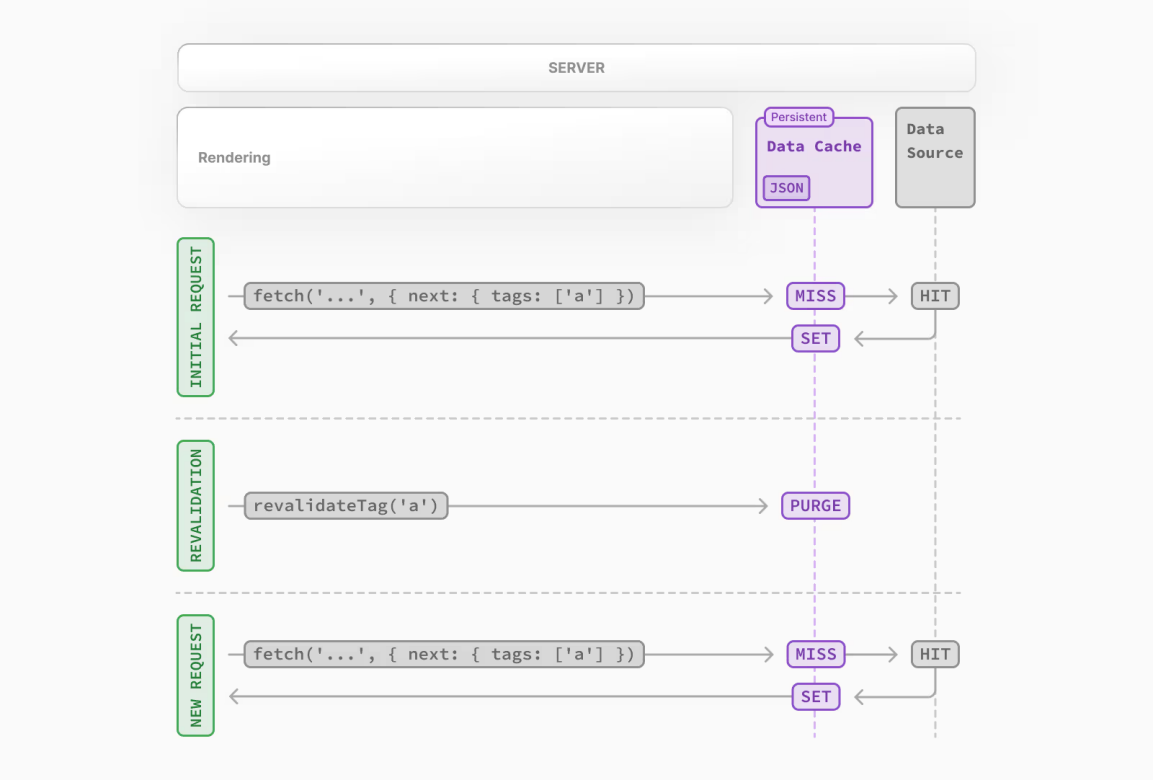
Any requests that are called within the specified timeframe (e.g. 60-seconds) will return the cached data.

After the timeframe, the next request will still return the cached (now stale) data.

- Next.js will trigger a revalidation of the data in the background.
- Once the data is fetched successfully, Next.js will update the Data Cache with the fresh data.
- If the background revalidation fails, the previous data will be kept unaltered.

On-demand Revalidation:

Data can be revalidated on-demand by path (revalidatePath) or by cache tag (revalidateTag).



The first time a fetch request is called, the data will be fetched from the external data source and stored in the Data Cache.

When an on-demand revalidation is triggered, the appropriate cache entries will be purged from the cache.

This is different from time-based revalidation, which keeps the stale data in the cache until the fresh data is fetched.

The next time a request is made, it will be a cache MISS again, and the data will be fetched from the external data source and stored in the Data Cache.

Opting out

If you do not want to cache the response from fetch, you can do the following:

```
let data = await fetch('https://api.vercel.app/blog',
                       {
                           cache: 'no-store'
                       });
```

Full Route Cache

Next.js automatically renders and caches routes at build time. This is an optimization that allows you to serve the cached route instead of rendering on the server for every request, resulting in faster page loads.

To understand how the Full Route Cache works, it's helpful to look at how React handles rendering, and how Next.js caches the result:

1. *React Rendering on the Server*

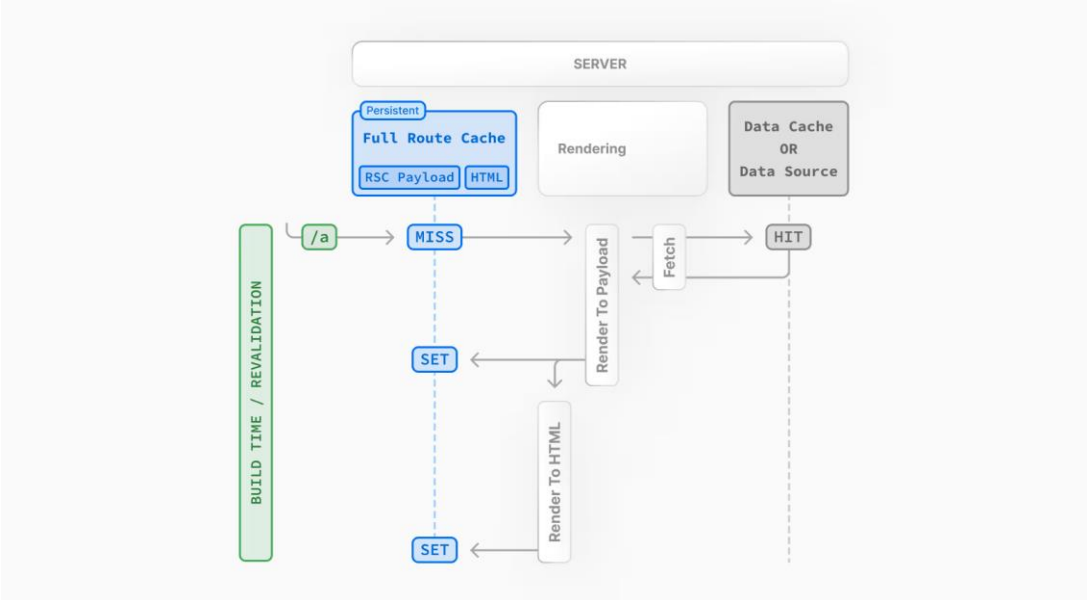
On the server, Next.js uses React's APIs to orchestrate rendering. The rendering work is split into chunks: by individual routes segments and Suspense boundaries.

Each chunk is rendered in two steps:

- React renders Server Components into a special data format, optimized for streaming, called the React Server Component Payload.
- Next.js uses the React Server Component Payload and Client Component JavaScript instructions to render HTML on the server.

This means we don't have to wait for everything to render before caching the work or sending a response. Instead, we can stream a response as work is completed.

2. *Next.js Caching on the Server (Full Route Cache)*



The default behavior of Next.js is to cache the rendered result (React Server Component Payload and HTML) of a route on the server. This applies to statically rendered routes at build time, or during revalidation.

3. *React Hydration and Reconciliation on the Client*

At request time, on the client:

- The HTML is used to immediately show a fast non-interactive initial preview of the Client and Server Components.
- The React Server Components Payload is used to reconcile the Client and rendered Server Component trees, and update the DOM.
- The JavaScript instructions are used to hydrate Client Components and make the application interactive.

4. Next.js Caching on the Client (Router Cache)

The React Server Component Payload is stored in the client-side Router Cache - a separate in-memory cache, split by individual route segment. This Router Cache is used to improve the navigation experience by storing previously visited routes and prefetching future routes.

5. Subsequent Navigations

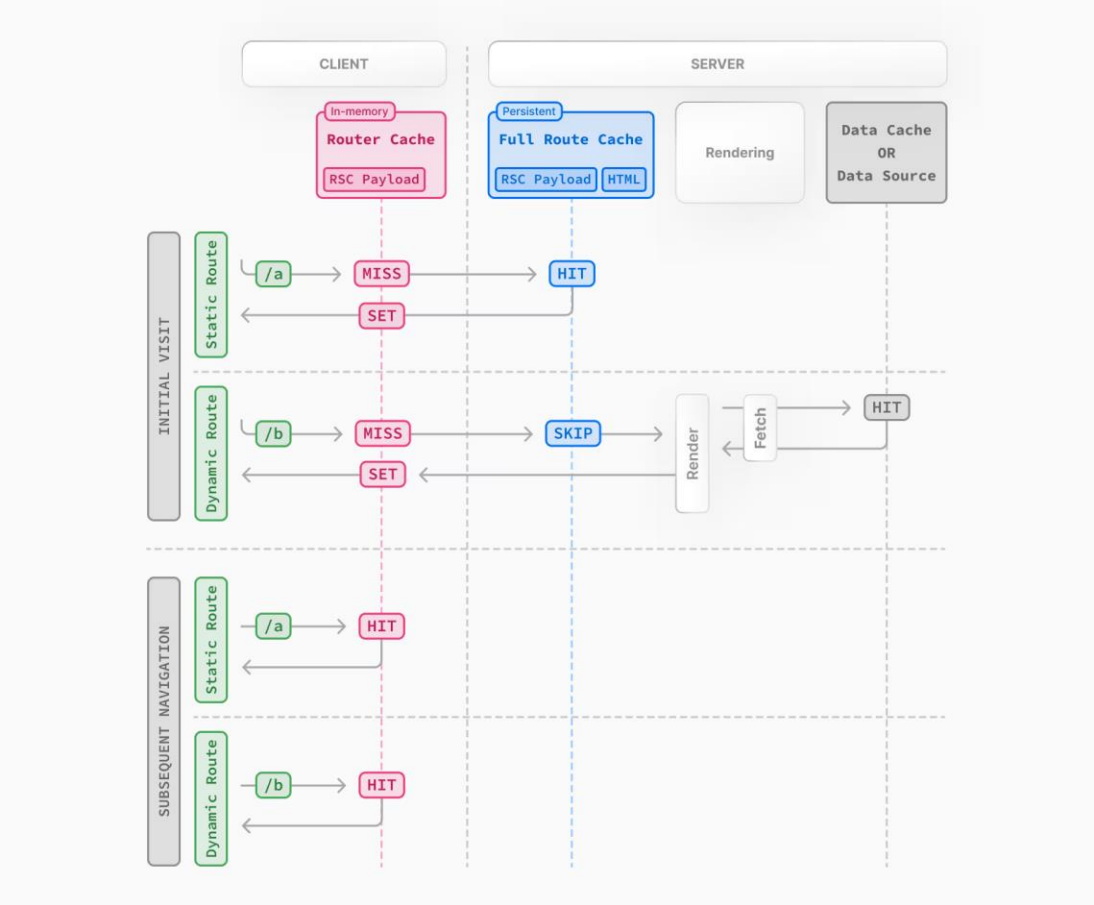
On subsequent navigations or during prefetching, Next.js will check if the React Server Components Payload is stored in the Router Cache. If so, it will skip sending a new request to the server.

If the route segments are not in the cache, Next.js will fetch the React Server Components Payload from the server, and populate the Router Cache on the client.

Static and Dynamic Rendering

Whether a route is cached or not at build time depends on whether it's statically or dynamically rendered. Static routes are cached by default, whereas dynamic routes are rendered at request time, and not cached.

This diagram shows the difference between statically and dynamically rendered routes, with cached and uncached data:



Duration

By default, the Full Route Cache is persistent. This means that the render output is cached across user requests.

Invalidation

There are two ways you can invalidate the Full Route Cache:

- Revalidating Data: Revalidating the Data Cache, will in turn invalidate the Router Cache by re-rendering components on the server and caching the new render output.
- Redeploying: Unlike the Data Cache, which persists across deployments, the Full Route Cache is cleared on new deployments.

Opting out

You can opt out of the Full Route Cache, or in other words, dynamically render components for every incoming request, by:

- Using a Dynamic API: This will opt the route out from the Full Route Cache and dynamically render it at request time. The Data Cache can still be used.
- Using the dynamic = 'force-dynamic' or revalidate = 0 route segment config options: This will skip the Full Route Cache and the Data Cache. Meaning components will be

rendered and data fetched on every incoming request to the server. The Router Cache will still apply as it's a client-side cache.

- Opting out of the Data Cache: If a route has a fetch request that is not cached, this will opt the route out of the Full Route Cache. The data for the specific fetch request will be fetched for every incoming request. Other fetch requests that do not opt out of caching will still be cached in the Data Cache. This allows for a hybrid of cached and uncached data.

Router Cache (Client-side)

Next.js has an in-memory client-side router cache that stores the RSC payload of route segments, split by layouts, loading states, and pages.

When a user navigates between routes, Next.js caches the visited route segments and prefetches the routes the user is likely to navigate to. This results in instant back/forward navigation, no full-page reload between navigations, and preservation of React state and browser state.

With the Router Cache:

- Layouts are cached and reused on navigation (partial rendering).
- Loading states are cached and reused on navigation for instant navigation.
- Pages are not cached by default, but are reused during browser backward and forward navigation. You can enable caching for page segments by using the experimental staleTimes config option.

Duration

The cache is stored in the browser's temporary memory. Two factors determine how long the router cache lasts:

- Session: The cache persists across navigation. However, it's cleared on page refresh.
- Automatic Invalidation Period: The cache of layouts and loading states is automatically invalidated after a specific time. The duration depends on how the resource was prefetched, and if the resource was statically generated:
    - Default Prefetching (prefetch={null} or unspecified): not cached for dynamic pages, 5 minutes for static pages.
    - Full Prefetching (prefetch={true} or router.prefetch): 5 minutes for both static & dynamic pages.

While a page refresh will clear all cached segments, the automatic invalidation period only affects the individual segment from the time it was prefetched.

Invalidation

There are two ways you can invalidate the Router Cache:

- In a Server Action:
    - Revalidating data on-demand by path with (revalidatePath) or by cache tag with (revalidateTag)
    - Using cookies.set or cookies.delete invalidates the Router Cache to prevent routes that use cookies from becoming stale (e.g. authentication).
- Calling router.refresh will invalidate the Router Cache and make a new request to the server for the current route.

Opting out

As of Next.js 15, page segments are opted out by default.

Cache Interactions

When configuring the different caching mechanisms, it's important to understand how they interact with each other:

Data Cache and Full Route Cache

- Revalidating or opting out of the Data Cache will invalidate the Full Route Cache, as the render output depends on data.
- Invalidating or opting out of the Full Route Cache does not affect the Data Cache. You can dynamically render a route that has both cached and uncached data. This is useful when most of your page uses cached data, but you have a few components that rely on data that needs to be fetched at request time. You can dynamically render without worrying about the performance impact of re-fetching all the data.

Data Cache and Client-side Router cache

- To immediately invalidate the Data Cache and Router cache, you can use revalidatePath or revalidateTag in a Server Action.
- Revalidating the Data Cache in a Route Handler will not immediately invalidate the Router Cache as the Route Handler isn't tied to a specific route. This means Router Cache will continue to serve the previous payload until a hard refresh, or the automatic invalidation period has elapsed.

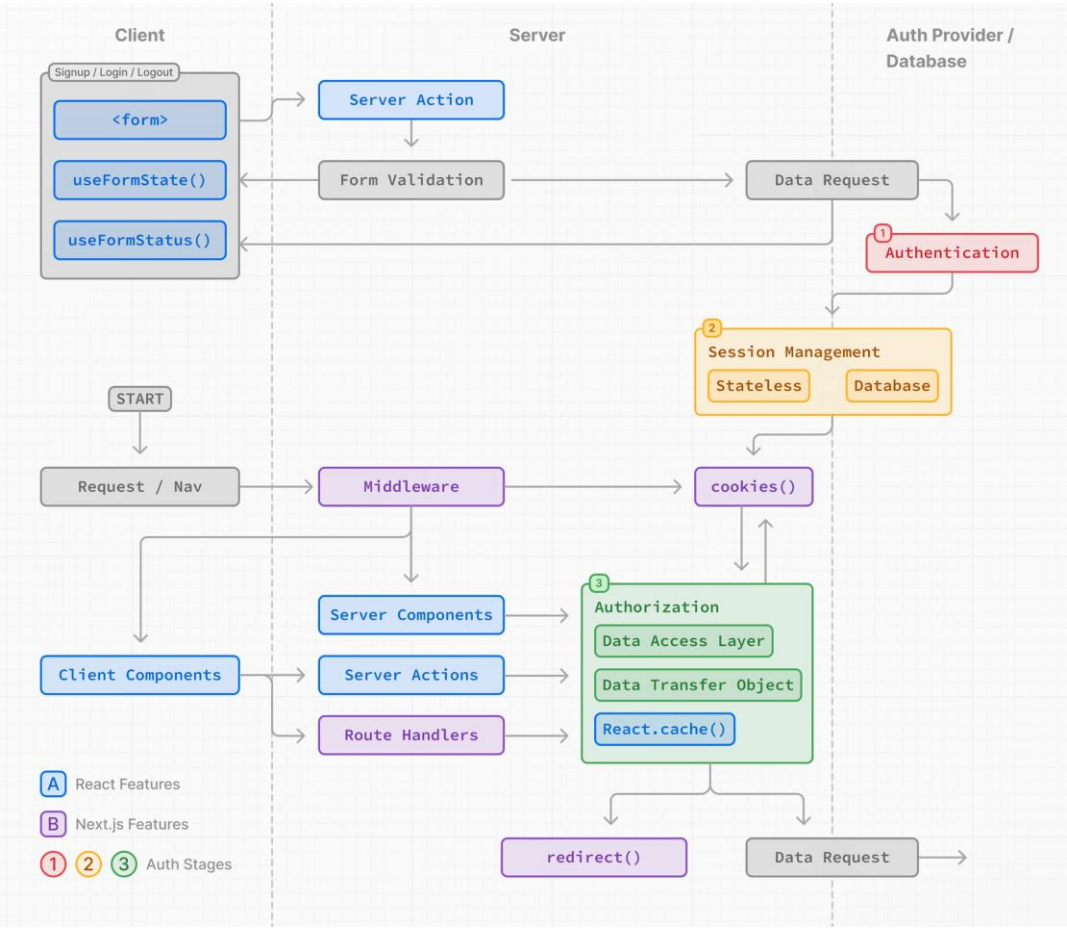
Authentication

Understanding authentication is crucial for protecting your application's data. This page will guide you through what React and Next.js features to use to implement auth.

Before starting, it helps to break down the process into three concepts:

1. Authentication: Verifies if the user is who they say they are. It requires the user to prove their identity with something they have, such as a username and password.
2. Session Management: Tracks the user's auth state across requests.
3. Authorization: Decides what routes and data the user can access.

This diagram shows the authentication flow using React and Next.js features:



The examples on this chapter walk through basic username and password auth for educational purposes. While you can implement a custom auth solution, for increased security and simplicity, we recommend using an authentication library. These offer built-in solutions for authentication, session management, and authorization, as well as additional features such as social logins, multi-factor authentication, and role-based access control.

Authentication

You can use the <form> element with React's Server Actions and useActionState to capture user credentials, validate form fields, and call your Authentication Provider's API or database.

Since Server Actions always execute on the server, they provide a secure environment for handling authentication logic.

Here are the steps to implement signup/login functionality:

1. Capture user credentials
   To capture user credentials, create a form that invokes a Server Action on submission. For example, a signup form that accepts the user's name, email, and password:

app/ui/signup-form.tsx

```
import { signup } from '@/app/actions/auth'

export function SignupForm() {
  return (
    <form action={signup}>
      <div>
        <label htmlFor="name">Name</label>
        <input id="name" name="name" placeholder="Name" />
      </div>
      <div>
        <label htmlFor="email">Email</label>
        <input id="email" name="email" type="email" placeholder="Email"
/>
      </div>
      <div>
        <label htmlFor="password">Password</label>
        <input id="password" name="password" type="password" />
      </div>
      <button type="submit">Sign Up</button>
    </form>
  )
}
```

2. Validate form fields on the server
   Use the Server Action to validate the form fields on the server. If your authentication provider doesn't provide form validation, you can use a schema validation library like Zod or Yup.
   Using Zod as an example, you can define a form schema with appropriate error messages:

app/lib/definitions.ts

```ts
import { z } from 'zod'

export const SignupFormSchema = z.object({
  name: z
    .string()
    .min(2, { message: 'Name must be at least 2 characters long.' })
    .trim(),
  email: z.string().email({ message: 'Please enter a valid email.'
}).trim(),
  password: z
    .string()
    .min(8, { message: 'Be at least 8 characters long' })
    .regex(/[a-zA-Z]/, { message: 'Contain at least one letter.' })
    .regex(/[0-9]/, { message: 'Contain at least one number.' })
    .regex(/[^a-zA-Z0-9]/, {
      message: 'Contain at least one special character.',
    })
    .trim(),
})

export type FormState =
  | {
      errors?: {
        name?: string[]
        email?: string[]
        password?: string[]
      }
      message?: string
    }
  | undefined
```

To prevent unnecessary calls to your authentication provider's API or database, you can return early in the Server Action if any form fields do not match the defined schema.

app/actions/auth.ts

```
import { SignupFormSchema, FormState } from '@/app/lib/definitions'

export async function signup(state: FormState, formData: FormData) {
  // Validate form fields
  const validatedFields = SignupFormSchema.safeParse({
    name: formData.get('name'),
    email: formData.get('email'),
    password: formData.get('password'),
  })

  // If any form fields are invalid, return early
  if (!validatedFields.success) {
    return {
      errors: validatedFields.error.flatten().fieldErrors,
    }
  }

  // Call the provider or db to create a user...
}
```

Back in your <SignupForm />, you can use React's useActionState hook to display validation errors while the form is submitting:

app/ui/signup-form.tsx

```tsx
'use client'

import { signup } from '@/app/actions/auth'
import { useActionState } from 'react'

export default function SignupForm() {
  const [state, action, pending] = useActionState(signup, undefined)

  return (
    <form action={action}>
      <div>
        <label htmlFor="name">Name</label>
        <input id="name" name="name" placeholder="Name" />
      </div>
      {state?.errors?.name && <p>{state.errors.name}</p>}

      <div>
        <label htmlFor="email">Email</label>
        <input id="email" name="email" placeholder="Email" />
      </div>
      {state?.errors?.email && <p>{state.errors.email}</p>}

      <div>
        <label htmlFor="password">Password</label>
        <input id="password" name="password" type="password" />
      </div>
      {state?.errors?.password && (
        <div>
          <p>Password must:</p>
          <ul>
            {state.errors.password.map((error) => (
              <li key={error}>- {error}</li>
            ))}
          </ul>
        </div>
      )}
      <button disabled={pending} type="submit">
        Sign Up
      </button>
    </form>
  )
}
```

3. Create a user or check user credentials
   After validating form fields, you can create a new user account or check if the user exists by calling your authentication provider's API or database.

Continuing from the previous example:

app/ui/signup-form.tsx

```
export async function signup(state: FormState, formData: FormData) {
    // 1. Validate form fields
    // ...

    // 2. Prepare data for insertion into database
    const { name, email, password } = validatedFields.data
    // e.g. Hash the user's password before storing it
    const hashedPassword = await bcrypt.hash(password, 10)

    // 3. Insert the user into the db or call an Auth Library's API
    const data = await db
      .insert(users)
      .values({
        name,
        email,
        password: hashedPassword,
      })
      .returning({ id: users.id })

    const user = data[0]

    if (!user) {
      return {
        message: 'An error occurred while creating your account.',
      }
    }

    // TODO:
    // 4. Create user session
    // 5. Redirect user
  }
```

After successfully creating the user account or verifying the user credentials, you can create a session to manage the user's auth state. Depending on your session management strategy, the session can be stored in a cookie or database, or both.

Session Management

Session management ensures that the user's authenticated state is preserved across requests. It involves creating, storing, refreshing, and deleting sessions or tokens.

There are two types of sessions:

1. **Stateless:** Session data (or a token) is stored in the browser's cookies. The cookie is sent with each request, allowing the session to be verified on the server. This method is simpler, but can be less secure if not implemented correctly.
2. **Database:** Session data is stored in a database, with the user's browser only receiving the encrypted session ID. This method is more secure, but can be complex and use more server resources.

Stateless Sessions

To create and manage stateless sessions, there are a few steps you need to follow:

1. Generate a secret key, which will be used to sign your session, and store it as an environment variable.
2. Write logic to encrypt/decrypt session data using a session management library.
3. Manage cookies using the Next.js cookies API.

In addition to the above, consider adding functionality to update (or refresh) the session when the user returns to the application, and delete the session when the user logs out.

1. Generating a secret key:
   There are a few ways you can generate secret key to sign your session. For example, you may choose to use the **openssl** command in your terminal:

   ```
   openssl rand -base64 32
   ```

   This command generates a 32-character random string that you can use as your secret key and store in your environment variables file:

   .env

   ```
   SESSION_SECRET=your_secret_key
   ```

   You can then reference this key in your session management logic:

   app/lib/session.js

   ```
   const secretKey = process.env.SESSION_SECRET
   ```

2. Encrypting and Decrypting sessions:
   Next, you can use your preferred session management library to encrypt and decrypt sessions. Continuing from the previous example, we'll use Jose (compatible with the Edge Runtime) and React's server-only package to ensure that your session management logic is only executed on the server.

app/lib/session.js

```
import 'server-only'
import { SignJWT, jwtVerify } from 'jose'
import { SessionPayload } from '@/app/lib/definitions'

const secretKey = process.env.SESSION_SECRET
const encodedKey = new TextEncoder().encode(secretKey)

export async function encrypt(payload: SessionPayload) {
  return new SignJWT(payload)
    .setProtectedHeader({ alg: 'HS256' })
    .setIssuedAt()
    .setExpirationTime('7d')
    .sign(encodedKey)
}

export async function decrypt(session: string | undefined = '') {
  try {
    const { payload } = await jwtVerify(session, encodedKey, {
      algorithms: ['HS256'],
    })
    return payload
  } catch (error) {
    console.log('Failed to verify session')
  }
}
```

3. Setting cookie (recommended option):
   To store the session in a cookie, use the Next.js cookies API. The cookie should be set on the server, and include the recommended options:
   - HttpOnly: Prevents client-side JavaScript from accessing the cookie.
   - Secure: Use https to send the cookie.
   - SameSite: Specify whether the cookie can be sent with cross-site requests.
   - Max-Age or Expires: Delete the cookie after a certain period.
   - Path: Define the URL path for the cookie.

app/lib/session.js

```
import 'server-only'
import { cookies } from 'next/headers'

export async function createSession(userId: string) {
  const expiresAt = new Date(Date.now() + 7 * 24 * 60 * 60 * 1000)
  const session = await encrypt({ userId, expiresAt })
  const cookieStore = await cookies()

  cookieStore.set('session', session, {
    httpOnly: true,
    secure: true,
    expires: expiresAt,
    sameSite: 'lax',
    path: '/',
  })
}
```

Back in your Server Action, you can invoke the createSession() function, and use the redirect() API to redirect the user to the appropriate page:

app/actions/auth.ts

```
import { createSession } from '@/app/lib/session'

export async function signup(state: FormState, formData: FormData) {
  // Previous steps:
  // 1. Validate form fields
  // 2. Prepare data for insertion into database
  // 3. Insert the user into the database or call an Library API

  // Current steps:
  // 4. Create user session
  await createSession(user.id)
  // 5. Redirect user
  redirect('/profile')
}
```

Updating (or refreshing) sessions:

You can also extend the session's expiration time. This is useful for keeping the user logged in after they access the application again. For example:

app/lib/session.ts

```
import 'server-only'
import { cookies } from 'next/headers'
import { decrypt } from '@/app/lib/session'

export async function updateSession() {
  const session = (await cookies()).get('session')?.value
  const payload = await decrypt(session)

  if (!session || !payload) {
    return null
  }

  const expires = new Date(Date.now() + 7 * 24 * 60 * 60 * 1000)

  const cookieStore = await cookies()
  cookieStore.set('session', session, {
    httpOnly: true,
    secure: true,
    expires: expires,
    sameSite: 'lax',
    path: '/',
  })
}
```

Deleting the session:

To delete the session, you can delete the cookie:

app/lib/session.ts

```
import 'server-only'
import { cookies } from 'next/headers'

export async function deleteSession() {
  const cookieStore = await cookies()
  cookieStore.delete('session')
}
```

Then you can reuse the deleteSession() function in your application, for example, on logout:

app/actions/auth.ts

```
import { cookies } from 'next/headers'
import { deleteSession } from '@/app/lib/session'

export async function logout() {
  deleteSession()
  redirect('/login')
}
```

Database Sessions

To create and manage database sessions, you'll need to follow these steps:

1. Create a table in your database to store session and data (or check if your Auth Library handles this).
2. Implement functionality to insert, update, and delete sessions
3. Encrypt the session ID before storing it in the user's browser, and ensure the database and cookie stay in sync (this is optional, but recommended for optimistic auth checks in Middleware).

For example:

app/lib/session.ts

```ts
import cookies from 'next/headers'
import { db } from '@/app/lib/db'
import { encrypt } from '@/app/lib/session'

export async function createSession(id: number) {
  const expiresAt = new Date(Date.now() + 7 * 24 * 60 * 60 * 1000)

  // 1. Create a session in the database
  const data = await db
    .insert(sessions)
    .values({
      userId: id,
      expiresAt,
    })
    // Return the session ID
    .returning({ id: sessions.id })

  const sessionId = data[0].id

  // 2. Encrypt the session ID
  const session = await encrypt({ sessionId, expiresAt })

  // 3. Store the session in cookies for optimistic auth checks
  const cookieStore = await cookies()
  cookieStore.set('session', session, {
    httpOnly: true,
    secure: true,
    expires: expiresAt,
    sameSite: 'lax',
    path: '/',
  })
}
```

After implementing session management, you'll need to add authorization logic to control what users can access and do within your application.

Authorization

Once a user is authenticated and a session is created, you can implement authorization to control what the user can access and do within your application.

There are two main types of authorization checks:

- Optimistic: Checks if the user is authorized to access a route or perform an action using the session data stored in the cookie. These checks are useful for quick operations, such as showing/hiding UI elements or redirecting users based on permissions or roles.

- Secure: Checks if the user is authorized to access a route or perform an action using the session data stored in the database. These checks are more secure and are used for operations that require access to sensitive data or actions.

For both cases, we recommend:

- Creating a Data Access Layer to centralize your authorization logic
- Using Data Transfer Objects (DTO) to only return the necessary data
- Optionally use Middleware to perform optimistic checks.

Testing

In React and Next.js, there are a few different types of tests you can write, each with its own purpose and use cases. This page provides an overview of types and commonly used tools you can use to test your application.

Types of tests

- **Unit Testing** involves testing individual units (or blocks of code) in isolation. In React, a unit can be a single function, hook, or component.
    - Component Testing is a more focused version of unit testing where the primary subject of the tests is React components. This may involve testing how components are rendered, their interaction with props, and their behavior in response to user events.
    - Integration Testing involves testing how multiple units work together. This can be a combination of components, hooks, and functions.
- **End-to-End (E2E) Testing** involves testing user flows in an environment that simulates real user scenarios, like the browser. This means testing specific tasks (e.g. signup flow) in a production-like environment.
- **Snapshot Testing** involves capturing the rendered output of a component and saving it to a snapshot file. When tests run, the current rendered output of the component is compared against the saved snapshot. Changes in the snapshot are used to indicate unexpected changes in behavior.

Async Server Components

Since async Server Components are new to the React ecosystem, some tools do not fully support them. In the meantime, we recommend using End-to-End Testing over Unit Testing for async components.

Testing Tools

We are going to explore some testing tools like Vitest, Jest, Playwright and Cypress.

Vitest

Vite and React Testing Library are frequently used together for Unit Testing. This section will show you how to setup Vitest with Next.js and write your first tests.

You can use **create-next-app** with the Next.js **with-vitest** example to quickly get started:

```
npx create-next-app@latest --example with-vitest with-vitest-app
```

Or we can manually setup Vitest, install vitest and the following packages as dev dependencies:

```
# Using TypeScript
npm install -D vitest @vitejs/plugin-react jsdom @testing-library/react @testing-library/dom vite-tsconfig-paths

# Using JavaScript
npm install -D vitest @vitejs/plugin-react jsdom @testing-library/react @testing-library/dom
```

Create a vitest.config.mts|js file in the root of your project, and add the following options:

vitest.config.mts

```
import { defineConfig } from 'vitest/config'
import react from '@vitejs/plugin-react'
import tsconfigPaths from 'vite-tsconfig-paths'

export default defineConfig({
  plugins: [tsconfigPaths(), react()],
  test: {
    environment: 'jsdom',
  },
})
```

Then, add a test script to your package.json

package.json

```
  ...
  ...
  "scripts": {
    "dev": "next dev --turbopack",
    "build": "next build",
    "start": "next start",
    "lint": "next lint",
    "test": "vitest"
  },
  ...
  ...
```

For more information on configuring Vitest, please refer to the Vitest Configuration docs.

Creating your first Vitest Unit Test

Check that everything is working by creating a test to check if the <Page /> component successfully renders a heading:

app/page.tsx

```tsx
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/students">Students</Link>
    </div>
  )
}
```

__tests__/page.test.tsx (double underscores before and after tests)

```tsx
import { expect, test } from 'vitest'
import { render, screen } from '@testing-library/react'
import Page from '../app/page'

test('Page', () => {
  render(<Page />)
  expect(screen.getByRole('heading', { level: 1, name: 'Home'
})).toBeDefined()
})
```

The example above uses the common __tests__ convention, but test files can also be colocated inside the app router.

Now to run the test we just need to fire following command in terminal

```
npm run test
```

To explore Vitest in detail I will prefer exploring their official documentation, https://vitest.dev/guide/

Jest

Jest and React Testing Library are frequently used together for Unit Testing and Snapshot Testing. This guide will show you how to set up Jest with Next.js and write your first tests.

You can use create-next-app with the Next.js with-jest example to quickly get started:

```
npx create-next-app@latest --example with-jest with-jest-app
```

Or we can manually setup Jest, install jest and the following packages as dev dependencies:

```
npm install -D jest jest-environment-jsdom @testing-library/react @testing-library/dom @testing-library/jest-dom ts-node
```

After installing Jest, we can generate a basic Jest configuration file by running the following command:

```
npm init jest@latest
```

This will take you through a series of prompts to setup Jest for your project, including automatically creating a jest.config.ts|js file.

Update your config file to use next/jest. This transformer has all the necessary configuration options for Jest to work with Next.js:

jest.config.ts

```ts
import type { Config } from 'jest'
import nextJest from 'next/jest.js'

const createJestConfig = nextJest({
  // Provide the path to your Next.js app to load next.config.js and
.env files in your test environment
  dir: './',
})

// Add any custom config to be passed to Jest
const config: Config = {
  coverageProvider: 'v8',
  testEnvironment: 'jsdom',
  // Add more setup options before each test is run
  // setupFilesAfterEnv: ['<rootDir>/jest.setup.ts'],
}

// createJestConfig is exported this way to ensure that next/jest can
load the Next.js config which is async
export default createJestConfig(config)
```

Under the hood, next/jest is automatically configuring Jest for you, including:

- Setting up transform using the Next.js Compiler.
- Auto mocking stylesheets (.css, .module.css, and their scss variants), image imports and next/font.
- Loading .env (and all variants) into process.env.
- Ignoring node_modules from test resolving and transforms.
- Ignoring .next from test resolving.
- Loading next.config.js for flags that enable SWC transforms.

Creating your first test

Your project is now ready to run tests. Create a folder called __tests__ in your project's root directory.

For example, we can add a test to check if the <Page /> component successfully renders a heading:

app/page.tsx

```tsx
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/students">Students</Link>
    </div>
  )
}
```

__tests__/page.test.tsx (double underscores before and after tests)

```tsx
import '@testing-library/jest-dom'
import { render, screen } from '@testing-library/react'
import Page from '../app/page'

describe('Page', () => {
  it('renders a heading', () => {
    render(<Page />)

    const heading = screen.getByRole('heading', { level: 1 })

    expect(heading).toBeInTheDocument()
  })
})
```

Now to run the test we just need to fire following command in terminal

```
npm run test
```

To explore Jest in detail I will prefer exploring their official documentation, https://jestjs.io/docs/getting-started

Playwright

Playwright is a testing framework that lets you automate Chromium, Firefox, and WebKit with a single API. You can use it to write End-to-End (E2E) testing. This guide will show you how to set up Playwright with Next.js and write your first tests.

The fastest way to get started is to use create-next-app with the with-playwright example. This will create a Next.js project complete with Playwright configured.

```
npx create-next-app@latest --example with-playwright with-playwright-app
```

Or we can manually setup Playwright:

```
npm init playwright
```

This will take you through a series of prompts to setup and configure Playwright for your project, including adding a playwright.config.ts file.

Creating your first Playwright E2E test

We are going to create two components and setup tests on them.

app/page.tsx

```tsx
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

app/about/page.tsx

```tsx
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>About</h1>
      <Link href="/">Home</Link>
    </div>
  )
}
```

Then, add a test to verify that your navigation is working correctly:

tests/example.spec.ts

```ts
import { test, expect } from '@playwright/test'

test('should navigate to the about page', async ({ page }) => {
  // Start from the index page (the baseURL is set via the webServer in
the playwright.config.ts)
  await page.goto('http://localhost:3000/')

  // Find an element with the text 'About' and click on it
  await page.click('text=About')

  // The new URL should be "/about" (baseURL is used there)
  await expect(page).toHaveURL('http://localhost:3000/about')

  // The new page should contain an h1 with "About"
  await expect(page.locator('h1')).toContainText('About')
})
```

Running your Playwright tests

Playwright will simulate a user navigating your application using three browsers: Chromium, Firefox and Webkit, this requires your Next.js server to be running.

We recommend running your tests against your production code to more closely resemble how your application will behave.

Steps to run test:

1. npm run build
2. npm run start
3. npx playwright test

After this if you want to see complete report you can run “npx playwright show-report” command.

References

https://nextjs.org/docs

https://en.wikipedia.org/wiki/Next.js