JS PRO MASTERY

The Ultimate Next.js Ebook

Table Of Contents

CHAPTER 1

Birth

In the first chapter, we explore the evolution of JavaScript and web development frameworks. We discuss the significance of embracing new technologies and compare code snippets in different frameworks to highlight their benefits. We introduce Next.js as a framework built on React.js, addressing limitations and incorporating new features.

The chapter concludes with the recommendation to shift focus to Next.js for building modern web applications.

Birth

Not too long ago, in 2015, React.js entered the scene. However, even the journey of JavaScript in the IT industry hasn't been exceptionally long. Originally developed by Brenden Eich at Netscape in 1995, JavaScript gained significant popularity during the 2000s.

This was largely due to Google's ingenious utilization of JavaScript to introduce interactive and dynamic features for map exploration. Subsequently, developers were introduced to frameworks and libraries, including jQuery, Angular, Node.js, React.js, and, most recently, Next.js. These technologies have revolutionized the web development landscape, offering developers various capabilities and possibilities.

You might wonder why this information is relevant in the context. The significance lies in the fact that it highlights the timeless truth that "change is constant." As we continue to advance and evolve as a society, our tools and technologies will naturally progress alongside us.

We have no other option but to embrace and adapt to these changes. It serves as a reminder that our willingness to embrace new ideas and technologies is essential for growth and success in the ever-changing IT industry landscape.

These technologies and tools share a common purpose: to enhance work efficiency and improve performance. In this era, we can still use vanilla JavaScript or create websites using HTML and CSS without a doubt.

However, when it comes to developing applications on a large scale, the efficiency of using the latest technologies surpasses that of traditional approaches. To showcase and experiment with this concept, we have created a video on "How to create a website using HTML & CSS" on our YouTube channel. You can personally analyze the amount of code and the level of efficiency demonstrated in the video.

To provide a brief glimpse of the evolution in JavaScript coding practices, here is a well-known code snippet implemented in various frameworks, starting from the core JavaScript language itself — The Hello World:

Vanilla JavaScript

```
// HTML: <button id="btn">Click Me</button>

document.getElementById('btn')
  .addEventListener('click', function () {
    alert('Hello, World!');
  });
```

jQuery

```
// HTML: <button id="btn">Click Me</button>

$('#btn').click(function () {
  alert('Hello, World!');
});
```

Angular

```
<!-- HTML: <button (click)="showMessage()">Click Me</button> -->

import { Component } from '@angular/core';

@Component({
  selector: 'app-example',
  template: `<button (click)="showMessage()">Click Me</button>`
})
export class ExampleComponent {
  showMessage(): void {
    alert("Hello, World!");
  }
}
```

React.js

```
import React from 'react';

class ExampleComponent extends React.Component {
  showMessage() {
    alert('Hello, World!');
  }

  render() {
    return <button onClick={this.showMessage}>Click Me</button>;
  }
}
```

"Ah? From what I see, there's an increase in the amount of code being written. It appears to be in complete opposition to what was mentioned earlier" — Are you thinking the same?

If we look at it solely from this perspective, one would certainly feel that the original language and framework require less code.

However, it's important to consider the bigger picture. And that's what truly matters, doesn't it? In reality, we don't just build "Hello World" projects. We undertake more substantial projects that demand the utilization of various frameworks and tools to achieve the desired functionality and scalability.

We could have talked about the “big picture” of using React or even Angular over vanilla code, but that is not the primary focus of this eBook. However, it is worth mentioning a few foundational reasons why these new tools make development more efficient:

Architecture — React and Angular follow a Component-Based Architecture, encouraging code reusability. For instance, if you create a component like a Button, you can use it anywhere in the application as often as needed. This reusability enhances the maintainability and scalability of the application.

Virtual DOM — The Virtual DOM is a lightweight representation of the actual DOM. It facilitates efficient and optimized updates to the user interface, resulting in improved performance. Simply put, it tracks changes within the application and performs a "diffing" process by comparing the previous version of the virtual DOM with the new version. It identifies the differences and updates the real DOM accordingly.

Ecosystem & Community — Modern libraries like React.js have vibrant and active communities. This provides developers with abundant resources, extensive documentation, reusable code packages, bug fixes, and support.

... and many other libraries or framework-specific reasons that you can explore. To truly appreciate the impact, I would once again recommend visiting the YouTube videos we created, where you can experience firsthand what it takes to build a simple landing page using two different tools to measure the efficiency of these tools:

Build and Deploy a Sushi Website using HTML & CSS

Watch and Code Now ↗

Build and Deploy a Bank Website using React.js

Watch and Code Now ↗

But hey, where does Next.js come in the picture?

As mentioned earlier, as we continue to progress, technology also advances. jQuery addressed the limitations of vanilla JavaScript, and then React.js emerged to overcome the shortcomings and loopholes of jQuery. However, even React.js has its own challenges, which have now been addressed by another tool called Next.js.

It's a big misconception that Next.js is a new language or library. **No!**

Vercel, the team behind Next.js, embarked on a unique approach to develop a framework encompassing client-side (frontend) and server-side (backend) functionalities within a single application. Guillermo Rauch, the original creator of Next.js and the mastermind behind Socket.IO, began working on this idea in 2016.

Over the course of a year, they continuously added new features such as file-based routing, automatic code splitting, hybrid rendering, internationalization, image and font optimization, and many more.

The relentless dedication of the Vercel developers, coupled with their ability to transform diverse ideas into reality, has caught the attention of Meta (previously known as Facebook) — the creators of React.js. Meta now explicitly recommends that developers use Next.js as their primary tool instead of relying solely on React.js. It's an extraordinary achievement!

And that's how we developers now need to shift our focus to the latest and greatest version, Next.js 15, to build highly capable and production-ready applications. This exciting evolution opens up new possibilities and opportunities for creating advanced web applications.

Onto the next chapter...

CHAPTER 2

Introduction

In this chapter, we'll dive into Next.js, a flexible React framework. We'll explore its advantages over React.js, including simplified frontend development, reduced tooling time, and an easy learning curve. We'll also discuss how Next.js improves performance, enhances SEO, and keeps advancing with new features.

By the end, you'll grasp the importance of mastering Next.js and be prepared to embark on an exciting journey with this framework.

Introduction

Next.js — A flexible React Framework. But what does that mean?

In software development, a framework serves as a tool equipped with predefined rules and conventions that offer a structured approach to constructing an application. It provides an environment that outlines the overall architecture, design patterns, and workflows, allowing developers **to focus on implementing specific application logic rather than dealing with low-level design.**

Simply put, a framework provides pre-built solutions for common functionalities such as integrating databases, managing routing, handling authentication, and more.

So what sets Next.js apart from React.js? — Next.js introduces plenty of features and capabilities that we will dive into in the upcoming chapter in detail.

But what you need to understand is Next.js is essentially an extension of React.js, incorporating pre-built solutions, ready-to-use features, and some additional functionalities. In other words, Next.js is built on top of React, expanding its capabilities.

If you're already a React.js developer, the Next.js journey will be silky smooth. If you don't know React.js, you should familiarize yourself with some of the main foundations of React.js, i.e., How to create a component, state management, code structure, etc.

To help you learn faster, we have a crash course on React.js that covers all the important things and includes a project for you to practice and test your skills:

React.js Crash Course

Watch and Code Now ↗

But **Why should you use a React Framework — Next.js?**

React is constantly evolving and revolutionizing the way websites are built. It completely transforms your approach to designing and developing applications.

It begins by encouraging you to think about components, breaking down the user interface into small pieces of code. You describe states to introduce interactivity and establish connections between these various components, shaping the flow of your application.

Implementing the great React architecture requires deep integration between all parts of your application, and this is where frameworks come in:

Less tooling time

Every aspect of frontend development has seen innovation, from compiling, bundling, minifying, formatting, etc., to deploying and more.

With Next.js, we won't have to worry about configuring these tools, thus investing more time in writing React code.

We can focus more on business logic using open-source solutions for routing, data fetching, rendering, authentication, and more.

Easy Learning Curve

If you are familiar with React.js, you will discover Next.js is considerably simpler.

Next.js, built on the foundation of React.js, offers exceptional documentation that provides comprehensive and detailed information on the why and how of using Next.js. The "Introduction | Learn Next.js" guide developed by the Vercel team is regarded as one of the best resources for the learning experience.

The constant updates and help from the community make the development process even easier.

One of the key aspects of Next.js is that it is not just a Frontend React Framework but a Full Stack React Framework enabling you to write backend code alongside your frontend code.

How does it contribute to an "Easy Learning Curve"? Wouldn't it be another thing to learn?

Absolutely not.

The backend aspect of the code you'll be working with is much simpler than you might anticipate. There's no need to set up anything or configure any routes as we have to do for any traditional backend app.

In fact, the Vice President of Vercel, Lee Robinson, expressed the following viewpoint:

> 💡 Moving from React + Express + Webpack to a framework resulted in removing **20,000+ lines of code** and **30+ dependencies** – while improving HMR (Hot Module Reloading) from **1.3s to 131ms**.

If the backend and tooling aspects discussed here seem confusing, there's no need to worry. In the upcoming chapters, we will dive into a practical comparison of how things are done with React.js and Express.js, as well as how both can be accomplished within Next.js.

Improved Performance

Next.js offers built-in features like server-side rendering, static site generation, and automatic code splitting, which optimize application performance by enabling faster initial page loads, improving SEO, and enhancing the user experience.

However, it doesn't mean server-side capabilities are limited to Next.js alone. React has introduced a new concept called React Server Components, which allows rendering components on the server side.

So, why choose Next.js over using React alone?

The advantage lies in the convenience and productivity provided by Next.js. By utilizing Next.js, you can leverage the existing features of React without the need for extensive setup and configuration.

Next.js automates many aspects, allowing you to focus more on utilizing the features rather than dealing with infrastructure & boilerplate code.

This approach follows the principle of "Convention over Configuration," streamlining the development process and reducing the amount of code you need to write compared to implementing React Server Components independently.

SEO - Search Engine Optimization

Perhaps the most ignored and must topic in an application's life, and the only drawback of React.js.

The key difference lies in the rendering approach between Next.js and React.js.

Search engine crawlers are like busy visitors to websites. They come and ask for the content of pages. They explore the links on those pages, carefully examining and organizing them for ranking purposes. This is what they do every day. To do their job well, they need to be able to access the content of the website pages.

React.js renders everything on the client side, sending a minimal initial HTML response from the server. The server sends a minimal HTML file code and a JavaScript file that the browser executes to generate the HTML. This poses a challenge for search engine crawlers to access and understand the complete content of the page.

On the other hand, Next.js provides the option of Static Site Generation (SSG) or Server Side Rendering (SSR).

With SSG or SSR, the server sends the complete HTML file and minimal JavaScript code to render only the content requiring client-side interaction. This enables search engine crawlers to access easily and index every page of the Next.js website accurately.

But, now you might wonder, "Why should I prioritize SEO?"

SEO is essential for making your website visible and highly ranked in search engine (browser) results. When you focus on SEO, you get several benefits, like more people visiting your website, better user experience, increased trust and credibility, and an advantage over your competitors because your website shows up higher in search results.

Giving priority to SEO can greatly impact how well your website does and how many people find it online.

Always Advancing

Next.js, the ever-evolving framework, consistently introduces new features to simplify developers' lives. With over 7+ versions released last year, Next.js focuses on innovation and improvement for a better user experience. This is precisely what frameworks like Next.js aim to achieve, making development easier and more efficient.

On top of that, other technologies like Expo, used for building React Native projects, are also adopting Next.js's groundbreaking features.

Inspired by Next.js's file-based routing system, Expo developers have implemented a similar feature — Expo Router to improve the decade-old routing system in React Native.

Isn't that great? Master one feature and effortlessly utilize it across multiple platforms

However, the list of features provided by Next.js goes beyond what has been mentioned so far.

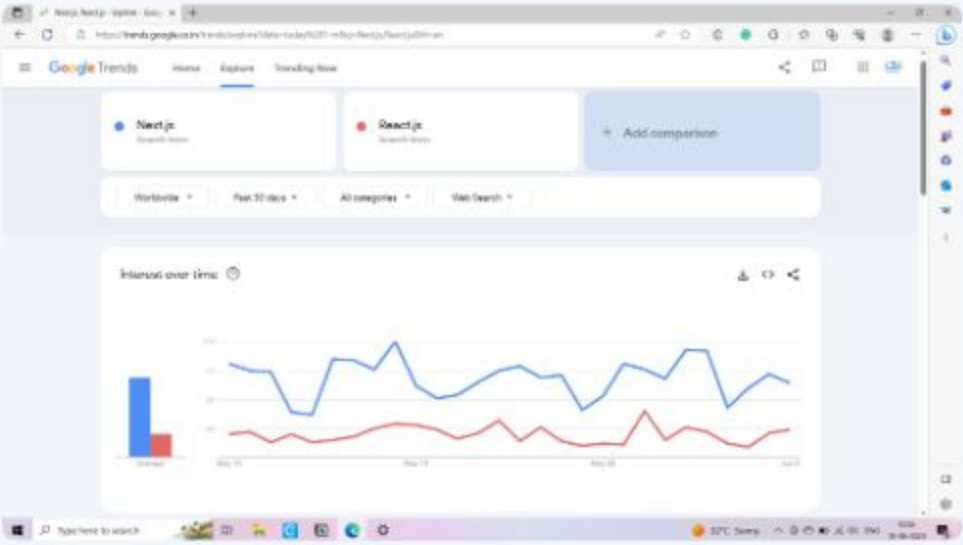
It offers a wide range of capabilities, including seamless file-based routing, efficient code splitting, image & font optimization, HMR(Hot Module Replacement), API Routes(backend), built-in support for Sass, CSS modules, data fetching choice (SSG, SSR, ISR), error handling, Metadata API (For SEO), Internationalization(support for any spoken language), etc.

It is best to try these features firsthand through practical implementation to truly appreciate its potential. That's precisely what we will do in the upcoming lessons – dive into the coding aspect!

"Hmm, alright. I'm willing to trust your insights on the new features of Next.js and such. However, is it actually being used by people? Are companies actively seeking professionals with Next.js expertise? Is there a high demand for it in the industry?"

— Are you wondering the same?

Let the data speak for itself:



In the past 30 days, Next.js has received significantly higher search interest worldwide than React.js.

But hey, that's just a Google trend. What about the industry? Are people even creating websites using Next.js?

Sure, let's take a look at "The Next.js Showcase" which shows different companies using Next.js:

- Notion
- Hulu
- Netflix Jobs
- Nike
- HBO Max
- Audible
- Typeform
- TED
- Auth0
- Product Hunt
- Hyundai
- Porsche
- repl.it
- Marvel
- Futurism
- Material-UI
- Coco Cola
- Ferrari
- Hashnode
- Verge

And many more renowned names. This demonstrates the genuine excitement and widespread adoption of Next.js!

Considering the rapid rate at which companies embrace Next.js, it would be no surprise to witness a huge surge in demand for Next.js jobs in the coming months, if not years.

Now is the perfect time to seize the opportunity and prepare for the future job market by mastering Next.js.

With this book and the courses we have done and will continue to do, you can be the next Next.js developer.

So, grab a cup of coffee, and let's get started on this exciting journey! 😀

CHAPTER 3

Roadmap

The Roadmap is a concise guide to web development essentials. It covers HTML for structuring web content, CSS for styling and layout, and JavaScript for interactivity. Learners will grasp important concepts like semantic tags, visual effects, variables, control flow, functions, and manipulating the DOM.

This chapter equips beginners with the skills needed to create dynamic and interactive web applications.

Roadmap

Before we start exploring Next.js, reviewing or relearning some basic concepts is a good idea to make learning easier. It all begins with building a solid foundation through fundamentals.

Think of this roadmap as a summary of what you should know as a foundation for learning Next.js. It's alright if you're unfamiliar with advanced topics like integrating databases or implementing authentication.

These points help you understand the main concepts without focusing on specific coding details.

In Next.js, there are various approaches to implementing these concepts. You have options like utilizing NextAuth (one of the coolest features), exploring popular market solutions like Clerk, or even building everything from scratch.

Similarly, when it comes to databases, you can choose between different options such as SQL databases like Postgres, NoSQL databases like MongoDB, or even consider using Prisma as an ORM (Object-Relational Mapping) manager.

Whether or not you have coding experience is not the most important factor here. What truly matters is understanding the underlying concepts. The roadmap is designed to introduce you to these concepts and familiarize you with the beneficial knowledge when aspiring to become a Next.js Developer.

Later in the next chapters, and with our branded courses, you'll learn how to do all the code stuff in Next.js. So don't worry; you have our back!

Presenting the Roadmap,

These points help you understand the main concepts without focusing on specific coding details.

1 Web Development Fundamentals

HTML - HyperText Markup Language

Basics

Understand the structure, elements, and attributes of HTML documents

- Structure
  - `<!DOCTYPE>` `<html>` `<head>` `<body>`
- Elements
  - Heading `<h1>` to `<h6>`
  - Paragraph `<p>`
  - Lists `<ul>` `<ol>` `<li>`
  - Link `<a>`
  - Image `<img>`
  - Input `<input>` `<textarea>`
  - Button `<button>`
  - Group Elements `<div>`

- Semantics

Use elements like

`<header>` `<nav>` `<main>` `<section>` `<aside>` `<footer>`

etc. to enhance document structure of accessibility.

- Forms

Learn to create forms, handling user input, perform form validations by using form element and onSubmit event listener

CSS — Cascading Style Sheets

Fundamentals

Understand the structure, elements, and attributes of HTML documents

- Structure
  - Box Model

    Understand how elements are styled using

    `padding` `margin` `border`

  - Selectors

    Learn about different types of selectors to target and style specific HTML elements. For example,

    `Type` `Class` `Id` `Child` `Sibling`

- Typography

  Explore text-related properties like

  `font` `size` `weight` `alignment`

- Colors and Backgrounds

  Understand how to set different

  `colors` `gradients` `background images`

Layout and Positioning

- Display

  Learn the various display values like

  `block` `inline` `inline-block`

- Position

  Explore how to position an element in different ways such as

  `relative` `absolute` `sticky` `fixed`

- Flexbox

  Master the flexbox layout to create responsive website layouts

- Grid

  Dive into CSS grid layout for advanced two-dimensional layouts

Effects

- Transitions

  Learn to create smooth transitions using different CSS properties like

  `delay` `timing` `duration` `property` `timing-function`

- Transformations

  Explore 2D and 3D transformations like

  `scaling` `rotating` `translating elements`

- Animations

  Learn how to create animations using keyframes

- Shadows and Gradients

  Explore with box shadows and linear or radial gradients

Advanced (Plus)

- Learn how to use CSS processors like sass or frameworks like tailwindcss for more powerful and efficient styling

JS JavaScript

- **Variables and Data Types**

  Declaring variables and understanding different data types such as string, number, boolean, null, undefined, object, array, etc.

- **Operators**

  Learn to use different operators such as arithmetic, comparison, logical, and assignment to perform operations on data

- **Control Flow**

  Try & test conditional statements such as if else, switch and loops such as for while to control program flow

- **Functions**

  Define and learn to create different functions, understand function scope, and work with different parameters and return values

- **DOM Manipulation**

  Knowing how to use JavaScript to change and interact with HTML elements on a webpage is an important skill. It's like a building block that we use in different ways with tools like React or Next.js in the form of new APIs.

  Remember when we showed you different "Hello World" examples in vanilla JavaScript and React.js? The basic idea is the same, but the code structure is a bit different in React.

If you're uncertain about how to learn & create a website using HTML, CSS, and JavaScript, you can immediately build an attractive Sushi Website by simply following the right free course:

Build and Deploy a Sushi Website using HTML & CSS

Watch and Code Now ↗

2 Modern JavaScript

- ES6 Features
  - Arrow Functions

    In JavaScript, there are different kinds of functions. One type that you'll often come across is called the Arrow function. Many prefer using arrow functions because they are shorter & easier to write.

    If you take the time to understand the syntax and how arrow functions work, it will help you write shorter and more straightforward functions. This can make your code look cleaner and easier to read.

  - Destructuring

    A helpful concept that will come in handy when we have to extract values from arrays and objects

- **Spread Syntax**

  Allows to expand elements of an array or object into individual elements

- **Template Literals**

  One of the widely used. Using the back sticks `` ` ``, we can interpolate strings with placeholders & expressions

- **Modules**

  Learn how to import export code between files to organize the code

- Asynchronous Programming
  - **Promises**

    Gain an understanding of the concept of promises and why they are necessary. Learn about resolving and rejecting promises and utilizing `then` and `catch` for handling asynchronous operations.

  - **Async/Await**

    Explore the usage of `async/await` to write asynchronous code in a more synchronous way. This convention is widely adopted as an alternative to using `then` and `catch` for handling promises.

  - **Fetch API**

    Discover how to use the Fetch API in the browser to send HTTP requests and handle the resulting responses.

- **Axios (Plus)**

  Explore the popular third-party library, Axios, which simplifies the process of making HTTP requests compared to the standard Fetch API.

- Additional JavaScript Concepts
  - **Array Methods**

    Familiarize yourself with different widely used array methods to simplify the development process. For example,

    `map` `slice` `includes`

    `filter` `splice` `join`

    `reduce` `forEach` `reverse`

    and few others

  - **Error Handling**

    One of the crucial part of web development is to catch and display the errors properly. No user will like to see the complete red screen with a text — "Error on line 35". Not even us.

    Therefore, it is essential for every developer to cultivate the skill of error handling. Familiarize yourself with the try-catch-finally block, which enables you to capture errors and present them on the user interface using user-friendly and easily understandable language.

3 The Ecosystem

Before we proceed learning the libraries and frameworks like React.js and Next.js, we'll need some kind of config to setup these projects:

- Foundations

  Node.js

  A javascript runtime environment that allows us to run JavaScript code outside of the browser.

  It's needed to work with React or Next.js. Make sure to download it

  NPM — Node Package Manager

  A tool that manages third party packages. Using it, you can download different packages needed inside you project like Axios

- Bundlers and Compilers

  Webpack

  Bundlers like webpack or Parcel help combine our JavaScript files and other assets into a single bundle

  Babel

  Transpilers like Babel convert modern JavaScript code to a version that works in all browsers

- Version Control (Plus)

  Learning a version control system like Git is highly valuable for anyone on the path to becoming a developer.

   **Git** - version control system

   **GitHub** - a web based platform to manage git repositories in the cloud

4 React JS

From here, you're efficiently learning Next.js. The same concepts are as it is used in the Next.

- Fundamentals
  - Components

    Think in terms of components. Learn to break the code or the UI in small manageable components for reusability & maintainability

    There are two ways in which we can create a component i.e.,

    - Class Component
    - Function Component — Widely used

    Along with that, learn "What is JSX?" and the "Component Lifecycle".

JSX is syntax in React that allows you two write HTML code. You won't even feel like you're using something different. It looks like HTML but isn't.

- **State and Props**

  State

  Learn how to create and manage state — A small store that holds a particular data of the application

  Props

  Learn how to pass props (a piece of data) between components

- **Events**

  Learn how to handle user interactions, such as clicks and form submissions, using event handlers.

- **Conditional Rendering**

  Learn how to conditionally render components and render dynamic list of data using the mapping techniques

P.S., Don't forget to learn about the special "Key" prop when rendering the dynamic list with map method.

- Hooks & Router
  - **Hooks**

    After learning how to create functional component, the next challenge will be to understand how to use hooks.

    Hooks are special functions that allows us to manage state, handle side effects and improve the efficiency. Few famous hooks are,

    `useState` `useEffect` `useRef` `useContext` `useCallback` `useMemo`

  - **Router**

    Learn how to do client side routing by understanding concepts like

    `Routes` `Route parameters` `Nested routes`

    > 💡 React Router DOM is an independent package used to handle the routing in React application

  - **State Management**

    Understand different state management options in React such as built-in state management — Context API

    `Context API` `Redux Toolkit` `Zustland`

- **Style**

  Explore different approaches to styling React components, including

  `Inline styles` `CSS modules` `Sass` `TailwindCSS` `Material UI`

  CSS-in-JS libraries like

  `styled-components` `Emotion`

- Forms & HTTP Requests
  - **Forms**

    Learn to create form validation, handling form submission with or without using third party libraries like,

    `Formik` `React Hook Form`

  - **HTTP Requests**

    Learn how to make requests using libraries like Axios or the built-in Fetch API

If you want to learn the fundamentals, styling and how to do HTTP Requests in React, you can check out our FREE and highly popular Crash Course on YouTube:

React JS Full Course 2023 | Build an App and Master React in 1 Hour - YouTube

Watch and Code Now ↗

If you want to enhance your skills in state management using tools like Redux Toolkit, you can explore our professional-level course:

Ultimate ReactJS Course with Redux Toolkit & Framer Motion jsmastery.pro

Watch and Code Now ↗

Also, if you want to create websites with a modern and attractive design that will impress clients or potential employers, we suggest you check out this series. It teaches you how to build React websites using different styling techniques like TailwindCSS, Sass, and even pure CSS.

Build and Deploy a Fully Responsive Modern UI/UX Site in React JS - YouTube

Watch and Code Now ↗

5 Backend

Although it's not a must to know how to do the backend to become a Next.js developer, it'll be nice to have the skill to showcase the ability to do both and become a full-stack Next.js developer.

You can use the following steps to learn backend development in any technology stack you prefer. It can even be Python

- Basics
  - **HTTP Protocol**

    Understand the HyperText Transfer Protocol and its fundamental concepts

  - **APIs and REST**

    APIs

    Learn what is Application Programming Interface (API)

    Explore methods, protocols and data formats that applications can use to to exchange information

    REST

    Learn what is Representational State Transfer

    - HTTP Methods

      `GET` `POST` `PUT` `DELETE` `PATCH`

- Status Code

```
200 – ok
201 – created
404 – Not Found
500 – Internal server error
```

- HTTP Headers
- Request and Response
- Resource URI

- CRUD
  - Understand the concept of CRUD operations

```
C – Creating data (POST)
R – Reading data (GET)
U – Updating data (PUT/PATCH)
D – Deleting data (DELETE)
```

- Authentication and Authorization

  Understand the difference between Authentication and Authorization

- User Sessions
- JWT — JSON Web Token
- Cookies
- Permissions and Roles

- Database

  Familiarize yourself with databases in storing and managing application data

  - Relational Database

     MySQL

     PostgreSQL

  - NoSQL Database

     MongoDB

     Redis

- Deployment
  - Environments
    - Production
    - Development
    - Staging

- Hosting Platforms

`Vercel` `Netlify` `Firebase` `Render`

`Heroku` `AWS Amplify` `Railway`

- Advanced (plus)
  - CI/CD — Continuous Integration/Continuous Deployment
  - Docker

Building backend applications can be challenging, but you can acquire the necessary skills with sufficient practice. If you're interested in in-depth project tutorials that specifically teach backend development using Express and MongoDB—a highly popular stack—feel free to explore some of our free courses available on YouTube:

Full Stack MERN Project - Build and Deploy an App | YouTube

Watch and Code Now ↗

Build and Deploy a Full Stack MERN App With CRUD, Auth, Charts - YT

Watch and Code Now ↗

If you want to enhance your skills in state management using tools like Redux Toolkit, you can explore our professional-level course:

Build & Deploy a Full Stack MERN AI Image Generation App | DALL-E Clone - YT

Watch and Code Now ↗

And now, at last, we will dive into the Next.js roadmap. It may not be necessary, as the content of this book is organized in a manner where each chapter serves as a guiding milestone, and it's the only resource you need (alongside some Build and Deploy courses, of course) to master Next.js!

But still, for you,

6 The Next.js

- Fundamentals
  - Learn why we should use Next.js and its benefits
  - Master the basic fundamentals of web development & React.js

    `State` `Prop` `Components` `Module`

  - Familiarize yourself with the ecosystem

    `Node` `NPM/Yarn` `NPX`

  - Setup a next.js application using create next app
- Architecture

  Understand the architecture of a Next.js application including different files and directories i.e., app directory vs pages directory.

  Next, dive into the backbone of Next.js functionality by exploring two distinct rendering processes:

  `Client` `Server`

- File Based Routing

  Learn how to create different types of routes in Next.js

  `Simple route` `Nested` `Dynamic` `Parallel` `Intercepting`

- Style

  Next.js has built-in support for CSS processors like Sass to CSS modules. Try different types of styling with Next.js to find the one that best fits your application:

  `CSS modules` `Tailwind CSS` `Sass`

- Data Fetching

  You have the flexibility to choose between different types of rendering and data fetching methods for your application. These methods include:

  `SSG` – Static Site Generation

  `SSR` – Server Side Rendering

  `ISR` – Incremental Static Regeneration

  `CSR` – Client Side Rendering

  It's important to understand each of these concepts in detail to determine how and when to implement the most suitable strategy for your application.

- SEO and Metadata

  Learn how to use SEO strategies and leverage the use of Metadata API of Next.js

  Points to learn:

  `Static Metadata` `Dynamic Metadata` `File based Metadata`

- **Handling Errors, loading states and much more**

  The latest Next.js 15 app directory introduces various file conventions that facilitate effective error handling, loading state management, displaying not found pages, and even organizing layouts in a more structured manner.

  Learn,

  `error.js file` `loading.js file` `not-found.js file` `layout.js file`

- **Authentication**

  Implementing custom email/password or social authentication becomes hassle-free with NextAuth in Next.js. Few auth libraries you can use with Next.js to speed up the development process:

  `NextAuth` `Clerk`

- **API routes**

  Explore how to create API routes — the backend:

  - **Route Handlers**

    Create custom request handlers

    `Static Route Handlers` `Dynamic Route Handlers`

  - **Middleware**
  - **Supported HTTP Methods**
  - **NextResponse**
  - **CORS and Headers**

- Database

  Discover how to incorporate various types of databases into your Next.js application by utilizing API routes.

  `MongoDB` `Postgres` `Prisma`

If you're someone who prefers video content over reading, you'll find our best and most up-to-date Crash Course on Next.js 15 on YouTube very enjoyable. This course not only covers the fundamentals of Next.js but also guides you in building a Full Stack project with authentication, utilizing the latest features of Next.js 15:

Next.js 15 Full Course 2024 | Build & Deploy a Full Stack Application - YouTube

Watch and Code Now ↗

If you have a keen interest in learning how to implement complex filtering, pagination, and searching using server-side rendering (SSR) with Next.js, then you should check out this resource:

Build and Deploy a Modern Next.js App | TypeScript, Tailwind CSS - YouTube

Watch and Code Now ↗

Keep in mind that real progress happens when you actively do coding. So, grab coffee, find a quiet spot, and start coding to make things happen.

CHAPTER 4

How it works

In this chapter, we lay the foundation by understanding how the web works before diving into Next.js code. We explore the traditional approach of HTML, CSS, and JavaScript websites, where the server sends complete files to the client for each page request.

We also introduce the React way, where the server sends a minimal HTML file and a bundled JavaScript file, and React manipulates the virtual DOM for efficient rendering. Finally, we discuss the Next.js way.

How it works

You might be itching to start with Next.js code, right?

Although writing code is important, we must first build our foundations. It'll not just help you in clearing interviews but will also help in making sound decisions in your application.

If your why isn't clear, you'll have no idea what you're doing, and you'll blame it on Next.js by saying that it's an overrated piece of technology. That will only showcase your lack of knowledge. It's a foolproof recipe to amaze everyone with your impressive ignorance.

So, perfect your why and your how will come naturally.

Let's time-travel a bit to see how things were used to work with different technologies.

The vanilla — HTML, CSS, and JavaScript

Websites built using the web's fundamental elements, namely HTML, CSS, and JavaScript, function differently compared to the latest technologies.
When a user visits such a website, their browser (the client) sends a request to the server (another computer where the site is hosted) asking for the content to be displayed.

Traditionally, for each of these requests, the server responds by sending three files i.e., the HTML, CSS, and JavaScript (only if any JavaScript code is involved). The client's browser receives these files and begins by analyzing the HTML file. Then, it applies the styles from the CSS file and implements any user interaction, such as event handlers or dynamic behavior, specified in the JavaScript file on the webpage.

The client will send additional requests to the server if the website has multiple pages. In response, the server will send the three files containing the respective content needed to render each page.

What's the catch?

Processing

Most processing occurs on the client side, meaning the user's web browser is responsible for rendering the HTML page and executing any JavaScript code present.

However, if the website is complex and the user's device needs more capabilities, it can strain the browser and create a burden for it to handle.

Bandwidth

As the server sends complete files to the client for each page request, it increases bandwidth usage. This becomes particularly significant when dealing with complex websites containing numerous pages and video and audio clips scattered throughout the site.

Load Time

The initial page load time may be longer when compared to the latest technologies. This is due to the complete transfer of files for each request. Only after the server has sent all the necessary files and the browser has finished parsing everything will we be able to view the website's content.

The React way

This is where React comes in. It improved the development lifecycle by introducing components, virtual DOM concepts, and the client-server mechanism.

When you access a React website, the client's browser sends a request to the server for the webpage content. In response, the server sends a minimal HTML file, which serves as the entry point for the entire application, along with a bundled JavaScript file.

React initiates client-side rendering using this JavaScript file, manipulating the virtual DOM. Instead of directly modifying the actual DOM, React updates the virtual DOM and then applies only the necessary changes to the real DOM, resulting in the desired page display.

React utilizes its client-side routing library, React Router, to navigate to different pages within the React application. This library enables changing the route without a full server request, preventing page refreshes.

React Router re-renders the relevant components based on the new URL when a new route is triggered. If the new page requires fetching data from the server, the corresponding components initiate requests to retrieve the necessary data.

What's the catch?

Complexity

Building a React application can present greater complexity than traditional HTML, CSS, and JavaScript websites. It involves thinking in components, managing state and props, and working with the virtual DOM, which may require a learning curve for developers new to React.js.

Processing

Similar to the traditional approach, react primarily performs client-side rendering. It heavily relies on JavaScript for initial rendering and subsequent requests to update the user interface, which are all handled on the client's browser.

However, this reliance on client-side rendering can delay rendering and interactivity, particularly on devices with slower processors and limited resources.

SEO

Yes, if you recall, we previously touched upon a notable drawback of React compared to Next.js in the Introduction chapter.

The issue is that search engine crawlers might need help fully accessing the website's content since everything is handled through JavaScript and only rendered on the client side. As a result, it impacts the website's visibility in search engine results

The Next.js Way - A blend of both

Knowing the benefits and limitations of both techniques, Vercel developers allowed us to choose where to render the content, on the client or server.

Typically, when a user visits a Next.js site, the client sends the request to the server, which starts executing the React Components, generates HTML, CSS, and JavaScript files, and sends back the fully rendered HTML to the client as a response. This file includes initial content, fetched data, and React component markup, making the client render it immediately without waiting for JavaScript to download and execute.

But it doesn't mean we don't receive any JavaScript files. The server will still send the JavaScript code as needed for the user interaction. From here, Next.js takes over and performs client-side hydration

Have you ever encountered the issue of a hydration error where the user interface doesn't match what was originally rendered on the server?

Well, this is what it is about. Hydration is attaching JavaScript event handlers and interactivity to the pre-rendered HTML. And when the placeholders of React components i.e., div, form, span, don't match what's being rendered on the client side, you see that error.

This is what it is — The hot topic of web development i.e., SSR.
Server Side Rendering!

For subsequent requests, you have full control over where to render your page content i.e., either on the server side (SSR) or the client side (CSR).

In the following chapters, we'll talk in-depth about different types of server-side rendering along with client-side rendering and when and where to render what.

CHAPTER 5

Create a Next.js Application

In this chapter, you will learn how to create a Next.js application. You'll start by setting up the necessary tools like Node.js and Visual Studio Code. Then, you'll create a new Next.js project using the **`create-next-app`** command.

You'll explore the project structure and understand the purpose of important files and folders. Finally, you'll learn about the **`jsconfig.json`** and **`package-lock.json`** files and their significance in managing dependencies.

Create a Next.js Application

By now, you should fully understand how the websites load. Now it's time to learn how to create websites using Next.js.

Setting up a Next.js application can be done in various ways. However, before we dive into that, there are a few things we need to have in place to get started with Next.js. The first requirement is having Node.js 16.8 or a more recent version. It's worth noting that there is a common misconception that Node.js is a new programming language.

In reality, it's a JavaScript runtime that enables the execution of JavaScript code outside of a web browser.

If you haven't installed Node.js before, you can visit this link and start downloading it. The website will give you two options based on your operating system: LTS and Current. The LTS (Long Term Support) version is the most stable, while the Current version is like a "Work in Progress" that adds new features but may have some bugs.

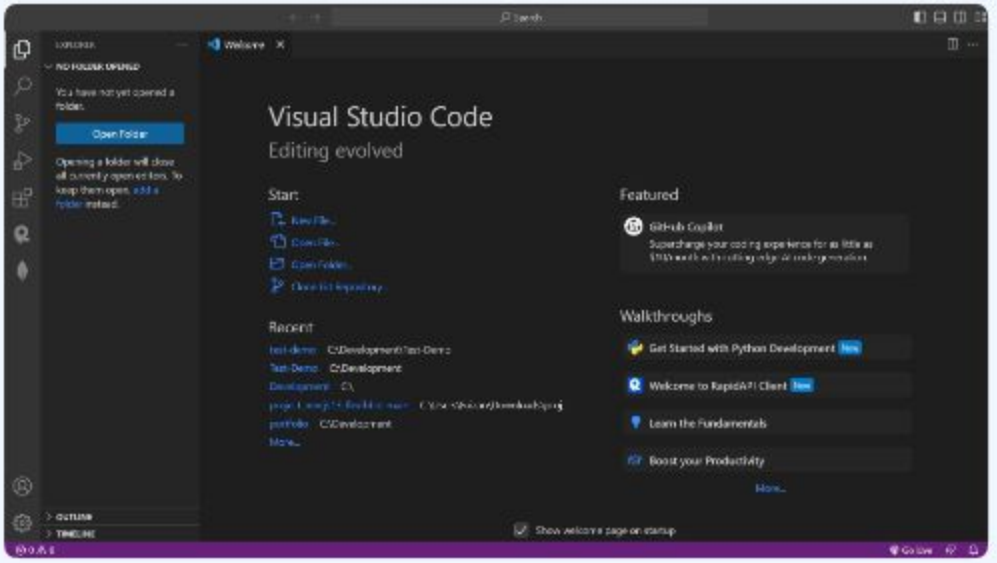
So, did you download it?

To determine the version you're using and check if you've downloaded Node.js, you can execute this command in your terminal or Command Prompt to verify:

```
node - v
```

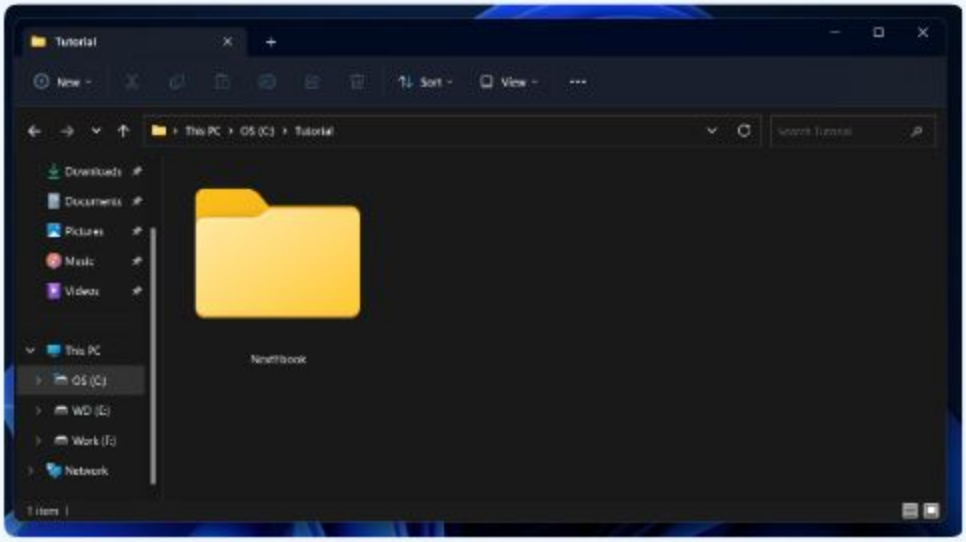
```
> node -v
18.16.1
>
```

Next, we require a Code Editor. Considering that Visual Studio Code (VSCode) is an exceptional editor, we suggest using it. You can visit this link, and depending on your operating system (OS), you will find the appropriate download link. The download process and installation process is as straightforward as it gets.



After downloading Node.js and VSCode, let's set up your first Next.js application.

Go to the desired location where you want to create your project. It can be any location, but it's advisable to maintain an organized structure. Create a new folder inside that location and name it "NextEbook." If you prefer a different name, feel free to choose one. This folder will hold all the code we will cover in this ebook.



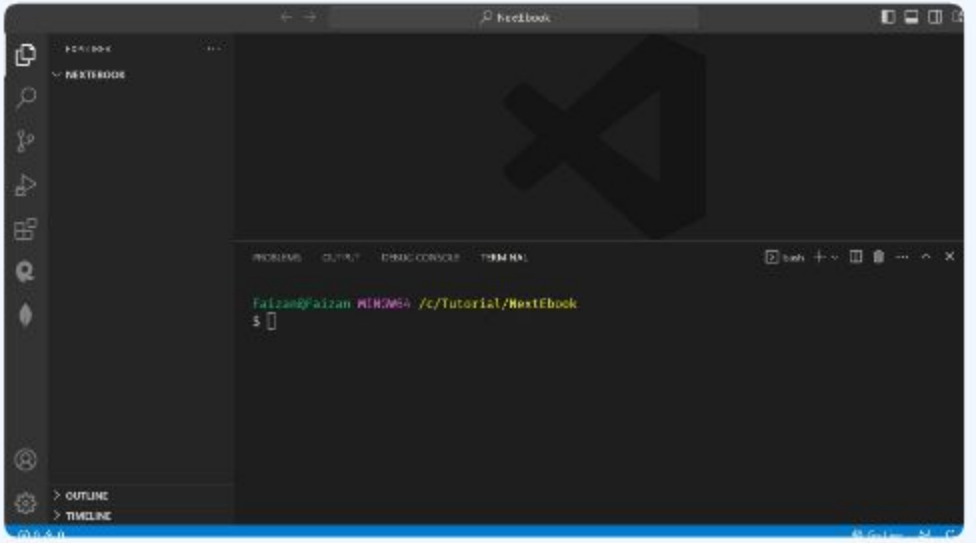
Now, let's proceed with the following steps to open the folder we just created in our chosen code editor, which is VSCode:

1. Launch VSCode.
2. Click on the "File" option in the top menu bar.
3. From the dropdown menu, choose "Open Folder."
4. Browse to the location where you created the "NextEbook" folder.
5. Select the "NextEbook" folder.
6. Click on the "Open" button.

Following these steps, you can view your "NextEbook" folder in VSCode.

VSCode provides its own built-in terminal, eliminating the need for developers to open the OS Terminal or Command Prompt separately. With the inline terminal in VSCode, we can perform all necessary tasks within the application.

To open the terminal, press **`ctrl + `(backtick)`** or click on the "Terminal" option in the top menu bar. From the dropdown menu, select "New Terminal." The terminal window will appear as follows:



For the final, let's now create our Next.js application. There are two options:

- Automatic Installation
- Manual Installation

As the name implies, manual installation involves obtaining and configuring packages individually and organizing the initial file and folder structure along with some code. We'll have to do everything on our own.

On the other hand, the alternative approach aims to accelerate the development process by allowing us to create the application with our preferred choices. Depending on our preferences, such as using TypeScript or not, incorporating styling libraries like Tailwind CSS, or opting for other options, we can set up the complete project with just one click.

Being widely used and an easy installation choice, we can create a next.js project by running a Zero Dependency CLI (Command Line Interface) tool — create-next-app. You can visit **this** link if you want to know about this in detail. Inside, you'll see how the create-next-app has been created with the templates, including with/without JavaScript, Tailwind CSS, etc.

Don't worry; you don't need to download create-next-app as another global package. Thanks to npx!

When you installed Node.js, you also got two other useful tools:

`NPM` `NPX`

NPM, which stands for Node Package Manager, allows us to manage and download various packages (collections of code) that we need to run our application. For example, packages like Axios for making HTTP requests or Redux Toolkit for state management.

On the other hand, NPX, short for Node Package eXecute, is a command-line tool. It lets us execute packages without installing them globally on our system. It's important to note that npx is used to run command-line tools and execute commands from packages, but it doesn't handle the installation of packages. That responsibility falls to npm, which takes care of package installation.

Let's move on from the theoretical discussion and proceed with the command that will automatically install the necessary packages for running a Next.js application.

```
npx create-next-app@latest
```

As soon as you press enter, it will prompt you to confirm whether it can download the required packages. Please select "yes" to proceed with the installation.

During the installation process, we will encounter a series of prompts individually. These prompts allow you to choose the specific features and configurations we desire for our Next.js application.

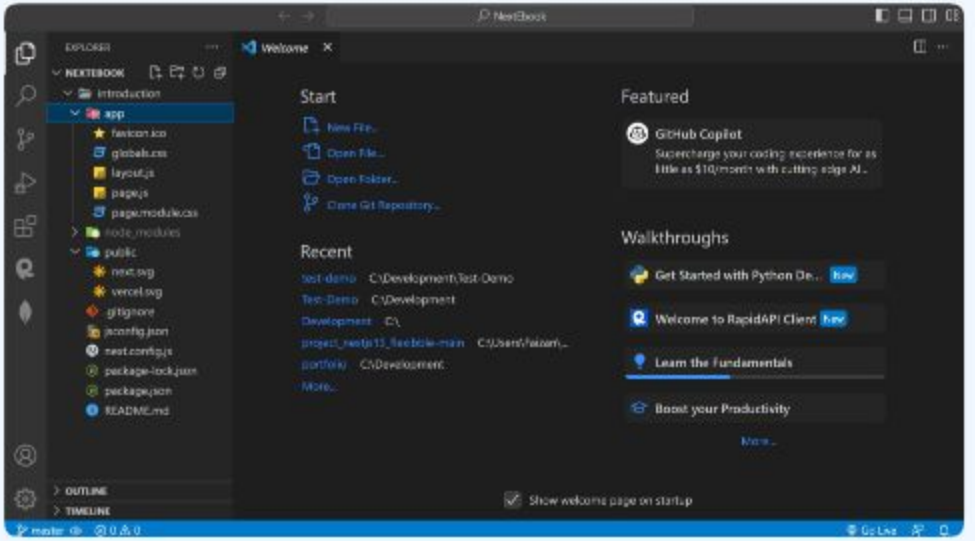
```
√ What is your project named? introduction
√ Would you like to use TypeScript with this project? No
√ Would you like to use ESLint with this project? No
√ Would you like to use Tailwind CSS with this project? No
√ Would you like to use src/ directory with this project? No
√ Use App Router (recommended)? Yes
√ Would you like to customize the default import alias? No
```

Let's choose not to include TypeScript, ESLint, and Tailwind CSS. In the upcoming chapters, we will explore these options in detail.

> If you see the installation process carefully, you'll see "Using npm." npx is used solely to execute commands from packages, while npm handles the installation of those packages.

And there you have it! The Next.js application has been successfully installed 🎉

Now, let's explore what's inside. Click on the "introduction" folder or the name you chose for your project. Inside, you will find several files and folders.

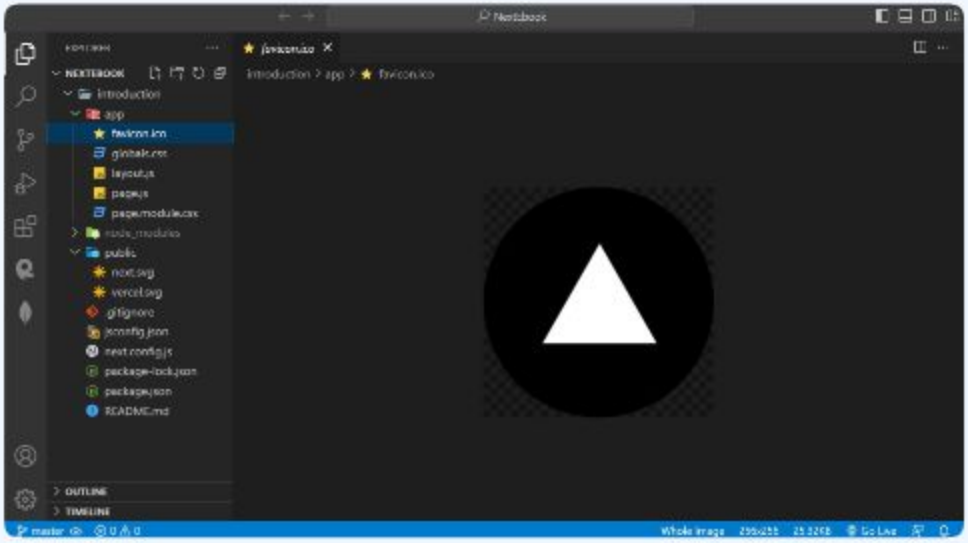


App

It's the root of the application, where we'll create various client (frontend) routes and potentially write backend code. Initially, you'll find some starter files in this location, including:

favicon.ico

It represents the website's favicon displayed in the browser's tab. By default, the favicon will be the Vercel icon. You can replace it with the one you like.



globals.css

This file holds the CSS code for the application. It is a global file where you can define CSS variables, import fonts, or perform other CSS initialization tasks.

You can keep the file, rename it, or even move it to a different location. It doesn't matter. However, if you make any changes to this file, you must update any other parts of the application that rely on it.

global.css

```
:root {
--max-width: 1100px;
--border-radius: 12px;
--font-mono: ui-monospace, Menlo, Monaco, 'Cascadia Mono',
'Segoe UI Mono','Roboto Mono', 'Oxygen Mono', 'Ubuntu Monospace',
'Source Code Pro','Fira Mono', 'Droid Sans Mono', monospace;

--foreground-rgb: 0, 0, 0;
--background-start-rgb: 214, 219, 220;
--background-end-rgb: 255, 255, 255;

--primary-glow: conic-gradient(
from 180deg at 50% 50%,
#16abff33 0deg,
#0885ff33 55deg,
#54d6ff33 120deg,
#0071ff33 160deg,
transparent 360deg
);
--secondary-glow: radial-gradient(
rgba(255, 255, 255, 1),
rgba(255, 255, 255, 0)
);.....
```

layout.js

It's the main entry point of the application. The root. The parent. The layout. Whatever you prefer to call it.

If you write anything in there, it'll appear on each & every client (frontend) route. It's universal.

If you need to import fonts, add metadata, wrap the application with Redux, or show a Navbar, this is the place to do it. All these tasks can be performed within this file.

page.js

It's an alias of the home route i.e., "/". It's important not to confuse this file with layout.js. Whatever you write inside page.js will be displayed only on the "/" route, while anything inside layout.js will appear across all routes, including the home route i.e., "/".

In short, layout.js is the parent of page.js, providing a common layout for all pages.

page.module.css

This CSS file is specifically designed to style a particular page component or module. The naming convention **`.module.css`** indicates that the CSS rules in this file are scoped to a specific component. In this case, it corresponds to the page.js component.

If you closely examine the code, you will notice the presence of the following:

```
import styles from './page.module.css`
```

inside the page.js file.

Node Modules

Well, it's the backbone!

The **`node_modules`** directory is a storage location for all the dependencies or packages required to run a project. Whenever we install packages using npm, the corresponding code for those packages is placed inside this directory.

For example, if we install Axios, a folder named **`axios`** will be created within the **`node_modules`** folder. If you've been following closely, you may have noticed that Next.js is built on top of React. Therefore, you can explore the **`node_modules`** folder to find the **`react`** folder, which contains the code for React itself.

In addition to React, you'll come across several other folders such as **`next`** (which enables Next.js-specific features), **`react-dom`**, **`postcss`**, **`styled-jsx`**, and more. Each of these folders contains numerous files and lines of code essential for running our Next.js application.

No need to worry, though. You don't have to interact with or visit this directory in the future (unless something terrible happens). npm automatically manages the **`node_modules`** folder when we install or uninstall node packages.

Public

It's a special folder allowing us to include static assets like images or files like fonts, videos, etc.

The content inside this folder is automatically optimized and available throughout the application. Thus it's advisable to put any PNGs, JPEGs, or SVGs we need inside this folder.

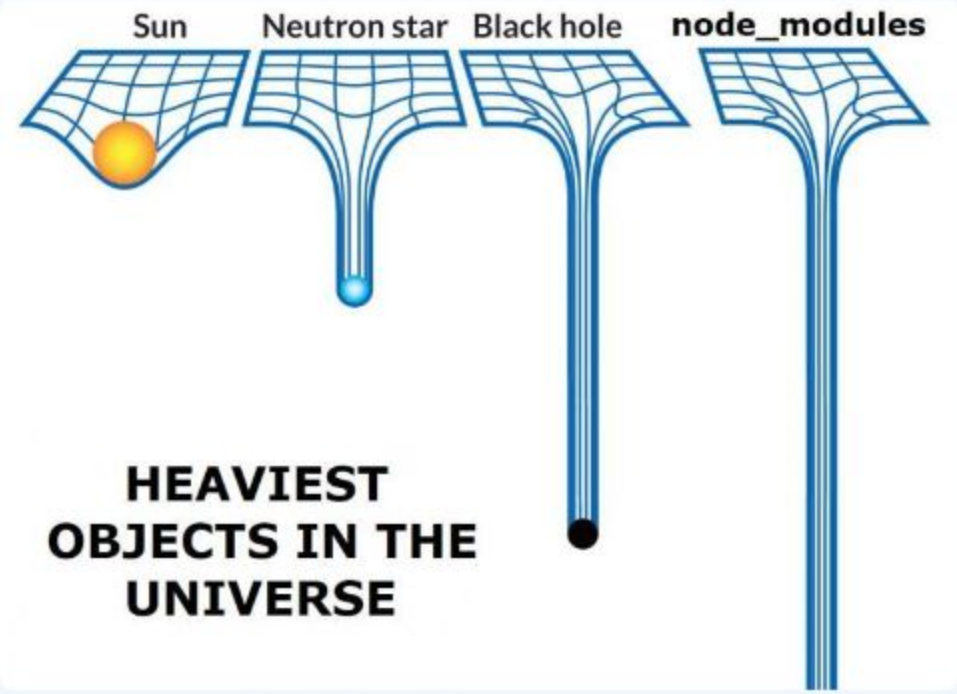
.gitignore

The **`.gitignore`** is a special text file that tells Git, the version control system, to exclude certain files and directories from being tracked.

One common entry in the **`.gitignore`** file is **`node_modules`**.

This is because the **`node_modules`** directory contains many files and folders generated when installing dependencies for a Node.js project. Including these files in version control would create unnecessary clutter and increase the size of the repository.

As said by the Creator of Node.js — Ryan Dahl



We would certainly not want to track the "Heaviest Object" in the universe!

jsconfig.json

Remember the prompt that appeared when we ran the Next.js CLI tool? It asked us if we wanted to customize the default import alias, and we chose "No".

```
√ Would you like to customize the default import alias? No
```

Well, the **jsconfig.json** file is related to that question. It act as a configuration file where we can set various options and settings for our project. One of the things we can configure is the import behavior.

By default, when we want to import code from one file into another, we use relative paths like this:

```
import something from '../../components'
```

This is the correct, but sometimes long and complex, relative path.

In larger projects or projects with complex directory structures, manually specifying relative paths for each import can become tedious and prone to errors.

However, by configuring the **compilerOptions** in the **jsconfig.json** file, we can inform the compiler about a specific import path structure that we want to use.

If we take a look at the content of the **jsconfig.json** file, we have:

```
{
    "compilerOptions": {
      "paths": {
      "@/*": ["./*"]
      }
    }
}
```

In this configuration, we have defined path aliases using `@/*`. This means that now we can do the following:

```
import something from '@/components'
```

i.e., using `@/*` for any files and folders located in this location `./*` i.e., the root!

Feel free to customize the alias to your preference. You can change `@/*` to `@*` or even `#/*` — the choice is yours! 😀

package-lock.json

Have you ever think of this as an unwanted file?

Well, let me tell you, it's actually quite important. This file, called **`package-lock.json`**, is automatically created when we install or make changes to packages. It serves as a lock file, carefully keeping track of the specific versions of all the installed packages and their dependencies.

Let's imagine a scenario: You're working on a Next.js project. After completing your work, you share the project code with your manager, but you omit the **`package-lock.json`** file, thinking it's not essential.

Now, your manager starts downloading the packages listed in the **`package.json`** file. The package manager, **`npm`**, installs the specified packages and their dependencies (i.e., the folder and files we have seen inside the **`node_modules`** folder).

However, if any dependencies release a new version, the package manager will eagerly download it. And unfortunately, if this new version may contain unresolved bugs or compatibility issues, it can cause your application to misbehave.

As a result, your manager may become frustrated, saying, "*It doesn't work on my machine* 😡," while you feel helpless, responding with a disheartened "*It works on my machine* 😢."

Know the importance and always share your **`package-lock.json`** file 😃

package.json

Think of this as an info card that tells about you — who you are, where you are from, etc. but with more complete details.

Along with containing the information regarding the name of the project, and its version, it tells us the dependencies and dev dependencies required to run this project

dependencies

List of the packages that are necessary for the project to run the production environment

devDependencies

List of packages that are only needed during the development process. For example, linting packages like eslint.

Whenever we install a package, whether a dependency or a dev dependency, npm automatically records the package name and its corresponding version in the respective section of the **`package.json`** file.

This way, the **package.json** file records all the packages required for the project, making it easier to manage and reproduce the development and production environments accurately.

Other than that, we can see another part i.e., "scripts". It contains executable commands or scripts using npm. We can completely customize it. Through these commands/scripts, we run tests, build a project, start a development server, or deploy the application.

Last but not least,

README.md

It's a like a manual or guidebook for your project — a markdown file where we can write down important information about our project, such as how to install it, what it does, and how others can contribute to it.

Having a good README helps people understand what our project is all about and how they can use it or get involved. It's like giving them directions or instructions on how to make the most of our project.

Now, let's finally run our application. We need to execute one of the commands from the "scripts" section we just mentioned, specifically the "dev" command.

Before proceeding, ensure the terminal's path is set to the correct location. Since we created a subfolder called "introduction" inside the "NextEbook" folder, we need to navigate into it. To do that, enter the following command in the open terminal:

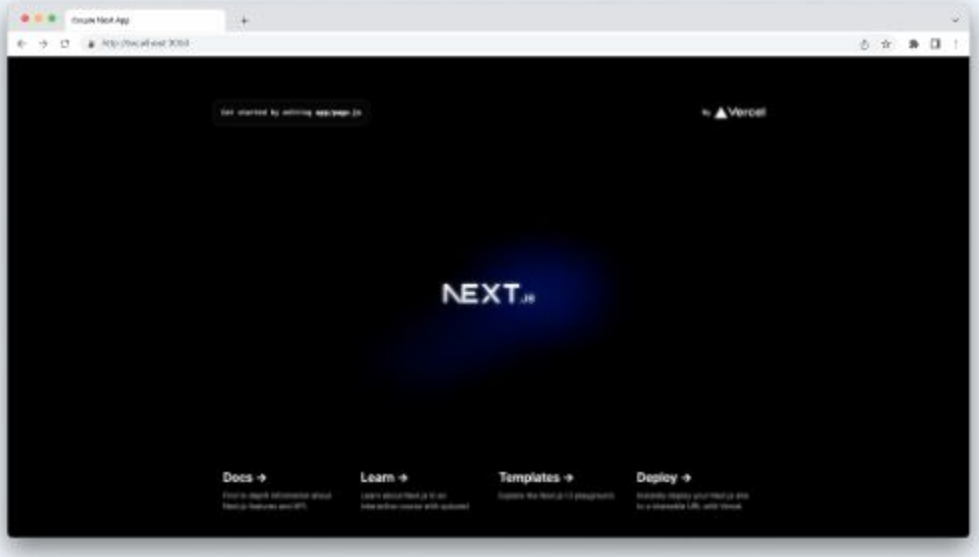
```
cd introduction
```

The "cd" command, which stands for "change directory," will navigate us inside the "introduction" folder.

Now run,

```
npm run dev
```

This command will start a local machine's development server on port 3000. To see the application in action, open your preferred web browser and type the URL: http://localhost:3000. If you have followed all the previous steps correctly, you should be able to see the application running as expected:



Phew, a lot of explaining just to cover the initial file and folder structure. But it's of no use if we don't take any actions. Let's change few things from the repo to see how it works.

Open the **page.js** file and delete all the existing code. We'll start fresh by creating the Home component.

```
export default function Home() {
  return (
    <main>
      <p>Hello World 👋</p>
    </man>
)}
```

After making the changes, save the file and return to your browser. If you visit the localhost again, you should see the updated content of "Hello World 👋" without manually refreshing the page. This is possible because of the Fast Refresh feature of Next.js, which automatically reflects the changes in real time as we edit the code.

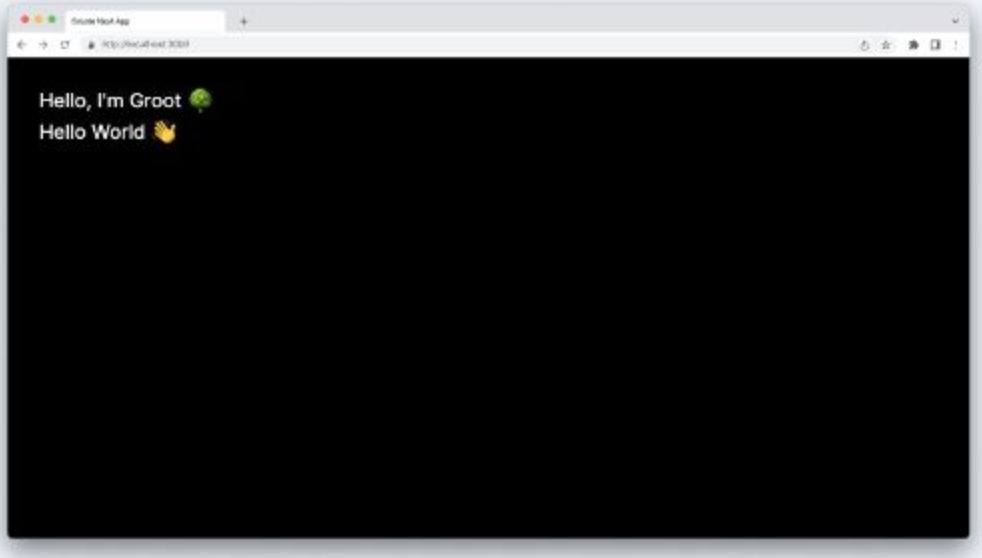
Now, open the **layout.js** file and add text inside the body tag:

```
import './globals.css';
import { Inter } from 'next/font/google';

const inter = Inter({ subsets: ['latin'] });

export const metadata = {
  title: 'Create Next App',
  description: 'Generated by create next app',
};

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body className={inter.className}>
        <p>Hello, I'm Groot 🌳</p>
        {children}
      </body>
    </html>
  );
}
```

Take a moment to ponder: Where will this text appear? Will it be displayed before the "Hello World 👋" or after it? Or perhaps it won't be displayed at all since we are on the home route, which is "/".

3, 2, 1... finished thinking?

To find the answer to your question, visit the browser. And you're right! The text will be displayed before the "Hello World 👋".



Why? Because **`layout.js`** is parent or root of the application.

But why before the "Hello World 👋"? Because we are rendering the children components of **`layout.js`** after the text "Hello, I'm Groot 🌳".

```
<body className={inter.className}>
   <p>Hello, I'm Groot 🌳</p>
   {children}
</body>
```

Reverse the order, and see the magic, i.e.:

```
<body className={inter.className}>
   {children}
   <p>Hello, I'm Groot 🌳</p>
</body>
```

The children prop is passed by Next.js to the RootLayout component in layout.js. It contains all the child components or route components of the application, starting from the home route and extending to any other routes that we may create.

```
// children is passed as a prop by Next.js
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body className={inter.className}>
        <p>Hello, I'm Groot 🌳</p>

        {/* Contains the route components, i.e., home route  */}
        {children}
      </body>
    </html>
  )
}
```

Clear enough?

Amazing! Before you rush to start the next chapter, I have a few tasks for you to complete:

Tasks 🎯

💻 Comment down the **{children}** inside the RootLayout and see if you can still see the "Hello World 👋"

```
import './globals.css'
import { Inter } from 'next/font/google'

const inter = Inter({ subsets: ['latin'] })

export const metadata = {
  title: 'Create Next App',
  description: 'Generated by create next app',
}

// children is passed as a prop by Next.js
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body className={inter.className}>
        <p>Hello, I'm Groot 🌳</p>

        {/* Contains the route components i.e., home route  */}
        {/* {children} */}
      </body>
    </html>
  )
}
```

❓ Explain the purpose of Node.js in the context of Next.js and web development.

❓ Explain the purpose and usage of the create-next-app CLI tool and why we use npx with it.

❓ What is the role of the **node_modules** directory in a Next.js project? Why is it recommended not to include it in version control?

CHAPTER 6

Client Vs. Server

In this chapter, you'll dive into the concepts of client-side rendering (CSR) and server-side rendering (SSR) in Next.js. You'll explore how Next.js handles rendering on the client and server, understand the benefits and trade-offs of each approach, and learn how to implement CSR and SSR in your Next.js applications.

Client Vs. Server

So far, we understand that Next.js does a mix of server-side and client-side rendering to get the best of both worlds. But we need to find out which parts of the application are rendered on the server side. Can we choose what to render in each environment?

And if so, how can we do that?

Before we answer these questions, let's go back and remind ourselves what we mean by client and server. What do these terms actually mean?

Client

The client refers to the device you are currently using, such as your smartphone or computer. The device sends requests to the server and displays the interface that we can interact with.

Server

The server is essentially just a computer device, but it is equipped with strong configurations and remains operational continuously. It is where all the code for our application is stored. When the client, our device, sends a request, the server performs the necessary computations and sends back the required outcome.

In previous versions of Next.js, specifically versions before 13, we faced a limitation where server-side rendering was restricted to individual pages. This meant that only the route pages like "**`/`**", "**`/about`**", "**`/projects`**", and so on could be rendered on the server side.

This limitation led to challenges such as prop drilling and duplication of API calls when passing data to lower-level components.

I recommend reading this article to gain a deeper understanding of the differences between the **pages** directory and the **app** directory in Next.js and how they address these limitations. It provides detailed insights into the topic:

Less code, better UX: Fetching data faster with the Next.js App Router

Link to blog ↗

And that, my friends, is where the app directory comes into action. It not only introduced many features but also brought about a revolutionary change, i.e., — Component level Server Side Rendering.

What does that mean?

It means that now we have the ability to choose where we want to render specific components or a section of code.

For instance, let's consider a component called **`Navbar.jsx`**. With this new capability, we can decide whether we want to render it on the server side or the client side (in the user's browser).

And that's how we end up with two types of components: Client Components and Server Components.

What are these?

Simply put, both are React components, but the difference lies in where they are rendered.

Client Component - A react component that runs/renders on the user's device, such as a web browser or mobile app.

Server Component - A react component that runs/renders on the server, i.e., the infra or place where we'll deploy our application

But why would we want to render such a small component on the server side?

Well, think about it!

By strategically deciding to render certain components on the server side, we save users' browsers from doing extra work with JavaScript to show those components. Instead, we get the initial HTML code for those components, which the browser can display immediately. This reduces the size of the JavaScript bundle, making the initial page load faster.

And as discussed above, we'll overcome our limitations with the **pages** directory. Rather than fetching and passing data to components separately, we can fetch the data directly within the component, turning it into a server component.

Overall, we'll have these benefits if we choose to do server-side rendering:

Smaller JavaScript bundle size: The size of the JavaScript code that needs to be downloaded by the browser is reduced.

Enhanced SEO (Search Engine Optimization): Server-side rendering helps improve search engine visibility and indexing of your website's content. (remember?)

Faster initial page load for better accessibility and user experience: Users can see the content more quickly, leading to a smoother browsing experience.

Efficient utilization of server resources: By fetching data closer to the server, the time required to retrieve data is reduced, resulting in improved performance.

Okay, but when to decide what to render where?

As their name suggests, where to render each component depends on what the component does.

If a component needs the user to interact with it, such as clicking buttons, entering information in input fields, triggering events, or using react hooks, then it should be a client component. These interactions rely on the capabilities of the user's web browser, so they need to be rendered on the client side.

On the other hand, if a component doesn't require any user interaction or involves tasks like fetching data from a server, displaying static content, or performing server-side computations, it can be a server component. These components can be rendered on the server without specific browser features.

Ask yourself:

To simplify things, the Next.js documentation provides a helpful table that guides you on where to render each component.

What do you need to do?	Server Component	Client Component
Fetch data.	✓	✗
Access backend resources (directly)	✓	✗
Keep sensitive information on the server (access tokens, API keys, etc)	✓	✗
Keep large dependencies on the server / Reduce client-side JavaScript	✓	✗
Add interactivity and event listeners (`onClick()`, `onChange()`, etc)	✗	✓
Use State and Lifecycle Effects (`useState()`, `useReducer()`, `useEffect()`, etc)	✗	✓
Use browser-only APIs	✗	✓
Use custom hooks that depend on state, effects, or browser-only APIs	✗	✓
Use React Class components	✗	✓

Impressive! You've just discovered one of the most significant features of the modern era. Take a well-deserved break!

Ready for more?

Another groundbreaking aspect of Next.js 15's **app** directory is that all components are automatically considered server components by default. That's right; every component is treated as a server component unless specified otherwise.

So how do we differentiate it from the client components?

Well, it's as simple as writing **`"use client"`** at the top of the component file. It's a straightforward and slick way to indicate that the component should be treated as a client component.

That's enough theory for now. Let's dive into coding 🚀

To get started, follow the steps outlined in the previous section to quickly set up a new Next.js project within the same **`NextEbook`** folder.

Before proceeding, ensure that the terminal's path points to the **`NextEbook`** folder and not the previously created **`introduction`** folder. To navigate out of the **`introduction`** folder, you can run the following command:

```
cd ..
```

Now, execute the **`create-next-app`** command. In this example, I'll be using the name **`client-server`** for the new folder, but feel free to choose any name you prefer.

Once the installation is complete, use the **cd** command to navigate into the **client-server** folder and start the application.

Now go to the app folder, then to the page.js file, and delete all other content except for the main and h2 tags:

app/page.js

```
import styles from './page.module.css'

export default function Home() {
  console.log("Where do I render?")

  return (
    <main className={styles.main}>
      <h2>Welcome 👋</h2>
    </main>
  )
}
```

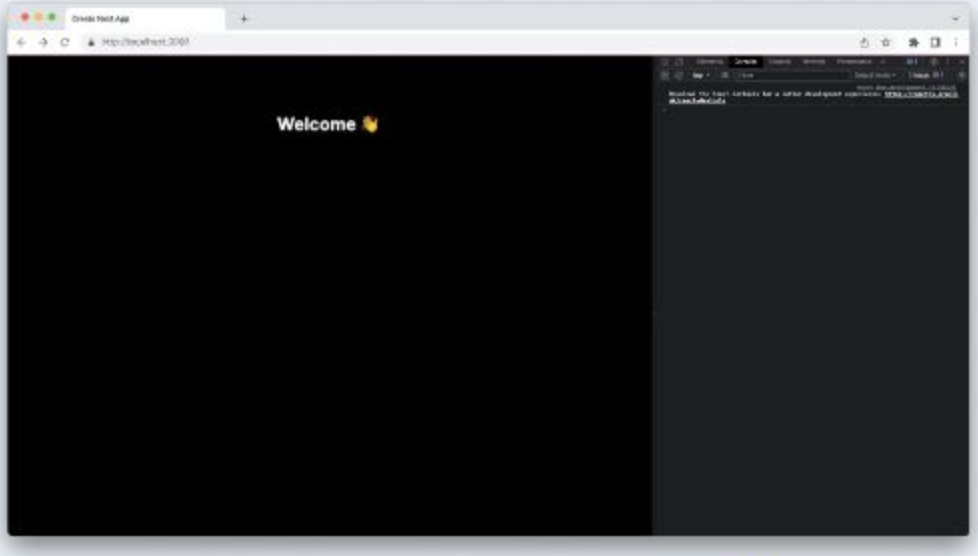
If you prefer, you can delete everything else from the **page.module.css** file and keep the styles only for the **main** tag of HTML.

app/page.module.css

```
.main {
  display: flex;
  flex-direction: column;
  align-items: center;
  padding: 6rem;
  min-height: 100vh;
}
```

After adding a console log in our page.js file, let's open our web browser to see if it appears there:

Once the installation is complete, use the **cd** command to navigate into the **client-server** folder and start the application.



Hmm, *"Where do I render?"* is not there. How on Earth? 🤔

Indeed, you are correct. As previously discussed, all these components will be Server Components by default!

So where do we see the console statements if not in the browser console? You know that, right?

The terminal! Let's return to our terminal and check if the mentioned log text is present there:

Terminal

```
> npm run dev

- next dev
- wait compiling /page (client and server)...
- event compiled client and server successfully in 390 ms (413 modules)

Where do I render?
```

And there we go, the log is there 😍

Now let's create two more components for each Client and Server.

Inside the root of the folder, i.e., outside of the **app** folder, create a new folder and name it **components**. Create two new files inside it, **ExampleClient.jsx** and **ExampleServer.jsx**.

components/ExampleClient.jsx

```jsx
"use client";

const ExampleClient = () => {
  console.log("I'm Client Component :)");

  return (
    <div>
      <p>This an example client component</p>
    </div>
  );
};

export default ExampleClient;
```

And the small ExampleServer component as

components/ExampleServer.jsx

```
const ExampleServer = () => {
  console.log("I'm Server Component :)");

  return (
    <div>
      <p>This is an example server component</p>
    </div>
  );
};

export default ExampleServer;
```

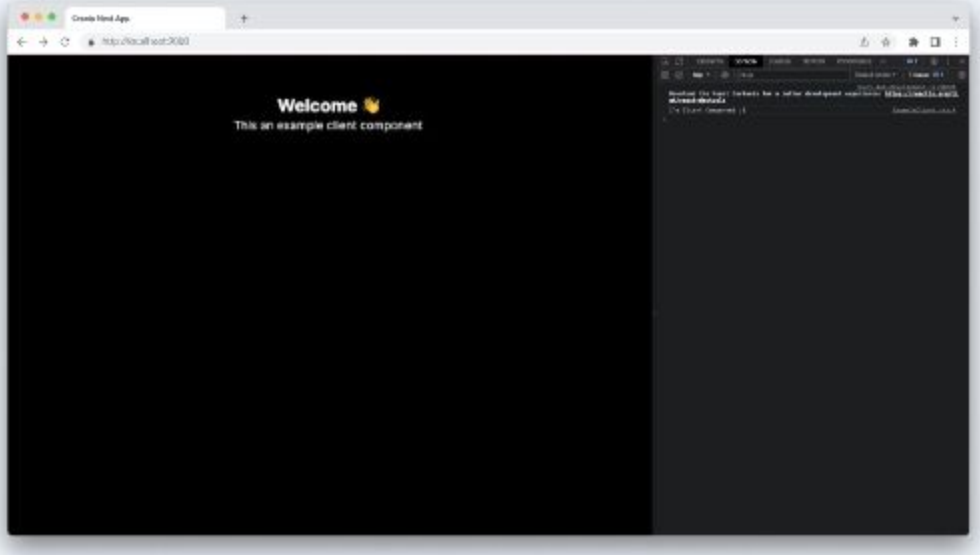
Now, first import & use the **ExampleClient** component inside the **app/page.js file**:

app/page.js

```
import styles from './page.module.css'
import ExampleClient from '@/components/ExampleClient'

export default function Home() {
  console.log("Where do I render?")

  return (
    <main className={styles.main}>
      <h2>Welcome 👋</h2>

      <ExampleClient />
    </main>
  )
}
```

Perfect. Let's check where we see which console log 😃

First browser,



Okay, that's right, right? We explicitly said Next.js to render **`ExampleClient.jsx`** as Client Component. Fair enough!

Going back to the Terminal, we see...

Terminal

```
> npm run dev
- next dev
- wait compiling /page (client and server)...
- event compiled client and server successfully in 390 ms (418 modules)
Where do I render?
I'm Client Component :)
```

Both of them, **why?**

This is because Next.js performs pre-rendering certain content before sending it back to the client.

So basically, two things happen:

- Server Components are guaranteed to be only rendered on the server
- On the other hand, client components are primarily rendered on the client side.

However, Next.js also pre-renders them on the server to ensure a smooth user experience and improve search engine optimization (SEO).

Next.js, by default, performs static rendering, which means it pre-renders the necessary content on the server before sending it to the client. This pre-rendering process includes server and client components that can be pre-rendered without compromising functionality.

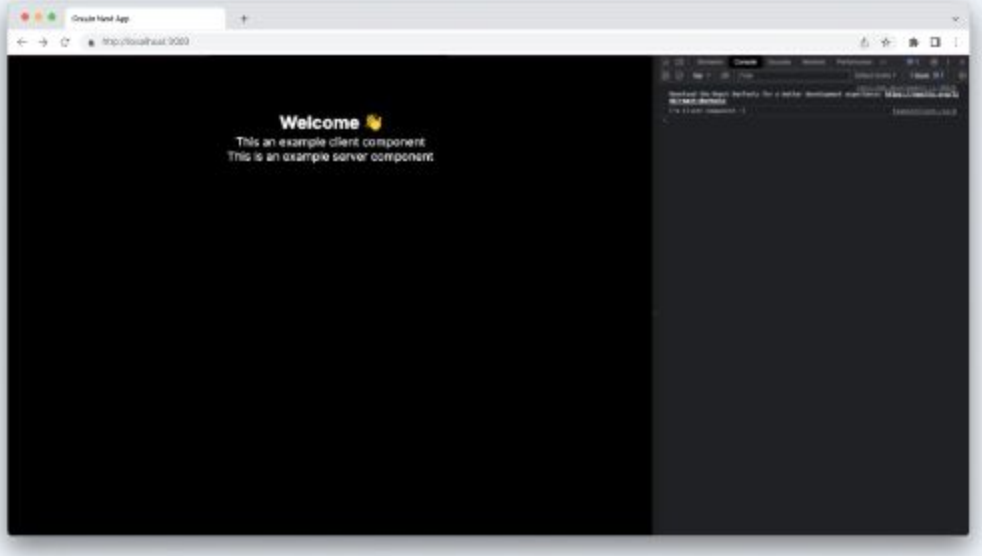
> The server Component is the latest React.js Feature. Next.js has simply used it over what they had, making the setup easy.

Let's play around with these components a little more. Now, import the **`ExampleServer`** component inside the **`app/page.js`** file

app/page.js

```
import styles from './page.module.css'
import ExampleClient from '@/components/ExampleClient'
import ExampleServer from '@/components/ExampleServer'

export default function Home() {
  console.log("Where do I render?")

  return (
    <main className={styles.main}>
      <h2>Welcome 👋</h2>

      <ExampleClient />
      <ExampleServer />
    </main>
  )
}
```

And now, if we visit the browser, along with showing both client-server component text on the website, it'll only show the "I'm Client Component :)" log inside the browser's console:



Whereas the terminal will show all the three console logs

Terminal

```
> npm run dev
- next dev
- wait compiling /page (client and server)...
- event compiled client and server successfully in 588 ms (548 modules)
Where do I render?
I'm Server Component :)
I'm Client Component :)
```

All good!

For the final play, let's remove the ExampleServer from **app/page.js** and add it inside the **components/ExampleClient.js**

app/page.js

```
import styles from './page.module.css'
import ExampleClient from '@/components/ExampleClient'

export default function Home() {
  console.log("Where do I render?")

  return (
    <main className={styles.main}>
      <h2>Welcome 👋</h2>

      <ExampleClient />
    </main>
  )
}
```

And the **ExampleClient** will look like this:

components/ExampleClient.jsx

```jsx
"use client";

import ExampleServer from "./ExampleServer";

const ExampleClient = () => {
  console.log("I'm Client Component :)");

  return (
    <>
      <div>
        <p>This an example client component</p>
      </div>
      <ExampleServer />
    </>
  );
};

export default ExampleClient;
```

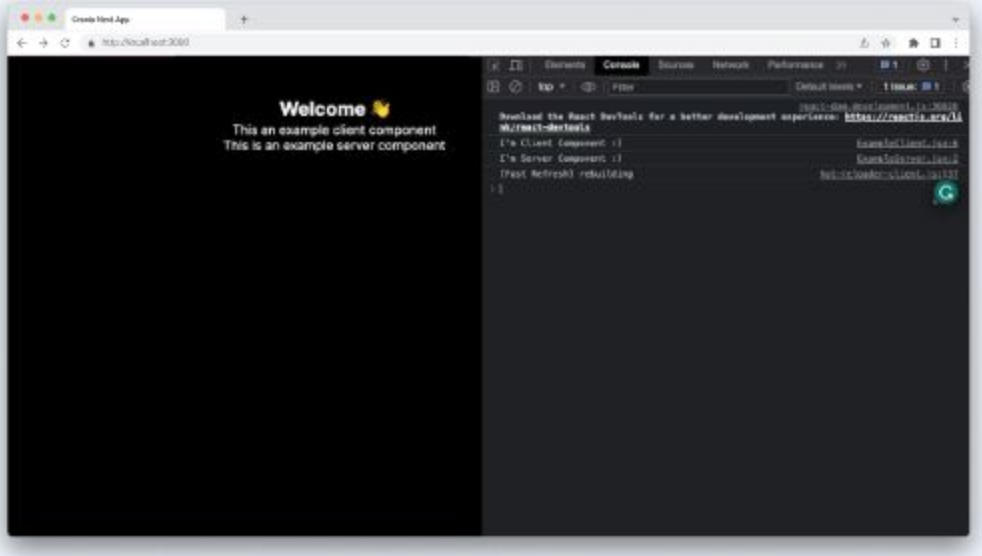
Hit save and see the result in both the Terminal and Browser console. First, let's see what the terminal shows:

Terminal

```
> npm run dev
- next dev
- wait compiling /page (client and server)...
- event compiled client and server successfully in 429 ms (549 modules)
Where do I render?
I'm Client Component :)
I'm Server Component :)
```

As expected, we see all three console logs due to the pre-rendering of Next.js and the server feature.

But something doesn't look good in the Browser console...



Why is the server component log appearing here? Wasn't it supposed to be on the server side only?

Well, in Next.js, there is a pattern at play. When we use "use client" in a file, all the other modules imported into that file, including child server components, are treated as part of the client module.

Consider "use client" as a dividing line between the server and client code. Once you set this boundary, everything inside it becomes client code.

Understood? If not, there is no need to worry.

Just remember: Do not include server components inside the client components.

And in case you encounter such a scenario, we have a solution. We'll discuss it in detail in its dedicated section, where we'll dive into the rendering process of client and server components. Additionally, we will share some valuable tips and tricks on creating different types of components depending on real-world examples.

Before we dive into yet another feature of Next.js, take some time out to work on the below tasks to solidify your learning so far:

Tasks 🎯

💻 Add "use client" inside the **`app/page.js`** file and see where the console logs are appearing:

app/page.js

```
"use client"

import styles from './page.module.css'
import ExampleClient from '@/components/ExampleClient'

export default function Home() {
  console.log("Where do I render?")

  return (
    <main className={styles.main}>
      <h2>Welcome 👋</h2>
      <ExampleClient />
    </main>
  )
}
```

- What are the different types of components in Next.js, and explain their difference?
- What are the benefits of server-side rendering?
- What are the latest features of the app directory regarding the client/server rendering?

CHAPTER 7

Routing

In this chapter, you'll learn about routing in Next.js and how it simplifies the process compared to React.js. Next.js uses a file-based router system, where folders and files define routes and UI components, respectively.

It covers creating a navigation bar component, organizing routes with folders, handling nested and dynamic routes, and leveraging Next.js' Fast Refresh feature. Overall, you'll gain a practical understanding of Next.js routing and its advantages.

Routing

Now, let's dive into routing!

One of Next.js's cool features is its ability to handle routes out of the box. But before we jump into that, let's first understand how routes are created in React.js.

Here's an example of how a route can be created using **`react-router-dom`** v6 in React.js:

App.js

```
import { BrowserRouter as Router, Route, Link, Routes } from 'react-
router-dom';

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </Router>
  );
};

export default App;
```

And here's a possible example of a nested dynamic routing for a multi-page website like an e-commerce site:

App.js

```
import React, { Fragment } from 'react';
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';

function App() {
  return (
    <Router>
      <Fragment>
        <Routes>
          <Route path="/" element={<Layout />}>
            <Route index element={<Home />} />
            <Route path="about" element={<About />} />
            <Route path="products" element={<Products />} />
            <Route path="products/:id" element={<ProductDetail />} />

            <Route path="seller" element={<SellerLayout />}>
              <Route index element={<Dashboard />} />
              <Route path="orders" element={<Orders />} />
              <Route path="reviews" element={<Reviews />} />
              <Route path="products" element={<SellerProducts />} />
              <Route path="products/:id" element={<SellerProductDetail />}>
                <Route index element={<SellerProductInfo />} />
                <Route path="pricing" element={<SellerProductPricing />} />
                <Route path="photos" element={<SellerProductPhotos />} />
              </Route>
            </Route>
          </Route>
        </Routes>
      </Fragment>
    </Router>
  );
}

export default App;
```

Scary, isn't it? Not only do we need to download and handle an external package, but as our application gets bigger, the routing becomes more complicated, making it harder to manage and understand!

Now, let's explore what Next.js brings to the table for routing

Next.js, aiming to simplify the process, uses a "file-based" router system.

Meaning,

- Folders are used to define routes
- Files are used to create UI for that route segment

For instance, to convert the previous React.js routing example into Next.js, we only need to create two folders named **`about`** and **`contact`**. We'll create a special file associated with that route segment inside each folder, such as **`page.js`** or **`page.jsx`**.

Let's understand it while we code. Quickly create a new Next.js app inside the NextEbook folder, like in the previous chapter.

I'll name the application **`routing`**. Before we begin, let's clean up the existing code to make it more organized and easier to understand for this use case:

- Remove **`page.module.css`** completely
- Add the CSS properties for the **`main`** tag inside the **`globals.css`** file. Leave the remaining CSS code unchanged.

app/globals.css

```css
main {
  padding: 2rem;
  display: flex;
  justify-content: center;
  align-items: center;
  flex-direction: column;
}
```

Remove the code inside **app/page.js** and keep only the main & h1 tag

app/page.js

```jsx
export default function Home() {
  return (
    <main>
      <h1>Home</h1>
    </main>
  )
}
```

Make sure everything is working correctly. Now, let's create a simple navigation bar (Navbar) component to move between the different pages we'll create easily.

Create a **components** folder inside the root of the application. Inside create two files **Navbar.jsx** and **navbar.module.css**

Navbar.jsx — contains a simple navbar with few links. We're using the **Link** component of **next/link** to navigate between the different routes of the application by providing the appropriate route path to the **href**.

components/Navbar.jsx

```jsx
import Link from 'next/link';
import styles from './navbar.module.css';

const Navbar = () => {
  return (
    <header>
      <nav className={styles.nav}>
        <p>Next.js</p>

        <ul className={styles.links}>
          <Link href="/">
            <li>Home 🏠</li>
          </Link>
          <Link href="/about">
            <li>About 👀</li>
          </Link>
          <Link href="/contact">
            <li>Contact 🏛️</li>
          </Link>
        </ul>
      </nav>
    </header>
  );
};

export default Navbar;
```

And create the corresponding style modules for the same i.e.,

navbar.module.css

components/navbar.module.css

```css
.nav {
  width: 100%;
  padding: 1.5rem 2rem;

  display: flex;
  align-items: center;
  justify-content: space-between;
  flex-wrap: wrap;

  border-bottom: 1px solid #2c2c2c;
}

.links {
  flex: 1;
  list-style: none;

  display: flex;
  justify-content: end;
  align-items: center;
  gap: 2rem;
}

.links li {
  white-space: nowrap;
  cursor: pointer;
}

.links li:hover,
.links li:active {
  color: #0099ff;
}
```

Nothing too complex. To ensure that the navbar appears on all route pages, we have two options:

1. The first option is importing the navbar component in each route page.
2. The second option is to import the navbar component in the parent component of these routes, such as layout.js.

Importing it into the parent component will consistently display the navbar across all route pages.

❌ Wrong Method

app/page.js

```
import Navbar from '@/components/Navbar'

export default function Home() {
  return (
    <main>
   <Navbar />
      <h1>Home</h1>
    </main>
  )
}
```

app/about/page.js

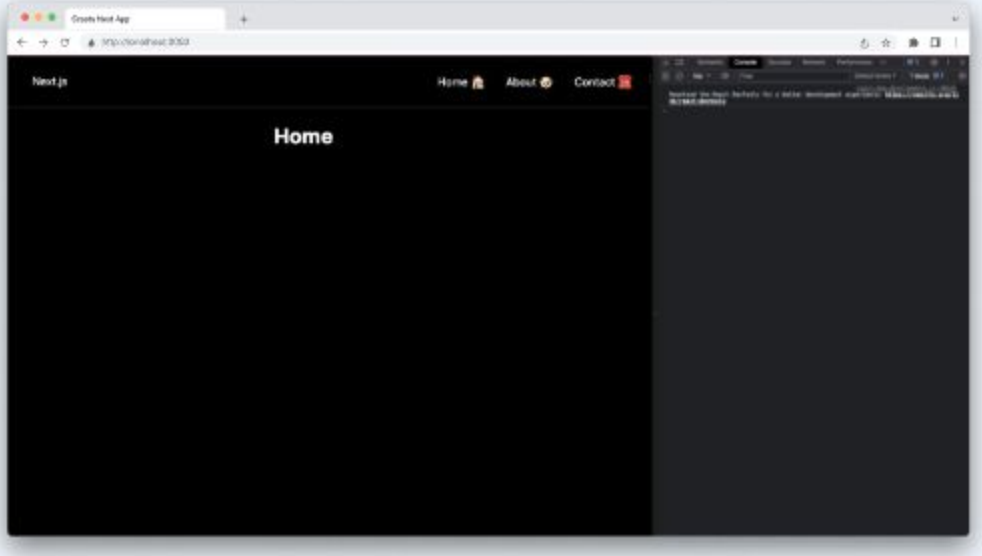
```
import Navbar from '@/components/Navbar'

export default function About() {
  return (
    <main>
   <Navbar />
      <h1>About</h1>
    </main>
  )
}
```

Right Method

app/layout.js

```
import './globals.css'
import { Inter } from 'next/font/google'

import Navbar from '@/components/Navbar'

const inter = Inter({ subsets: ['latin'] })

export const metadata = {
  title: 'Create Next App',
  description: 'Generated by create next app',
}

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body className={inter.className}>
        <Navbar />
        {children}
      </body>
    </html>
  )
}
```

By now, if you have followed all the steps properly, you should see this inside your browser:



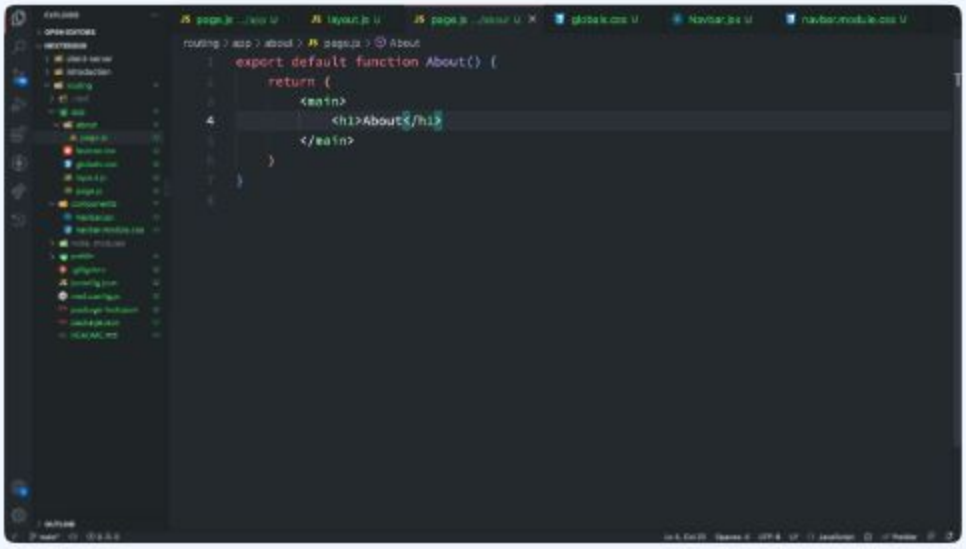
Now is the routing time!

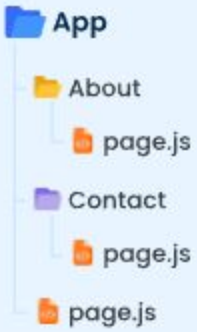
> Note: Please follow the Kebab Case writing convention when writing route names.

After creating the folder with the name about, create the special UI file page.js inside it to show the UI for that route:

app/about/page.js

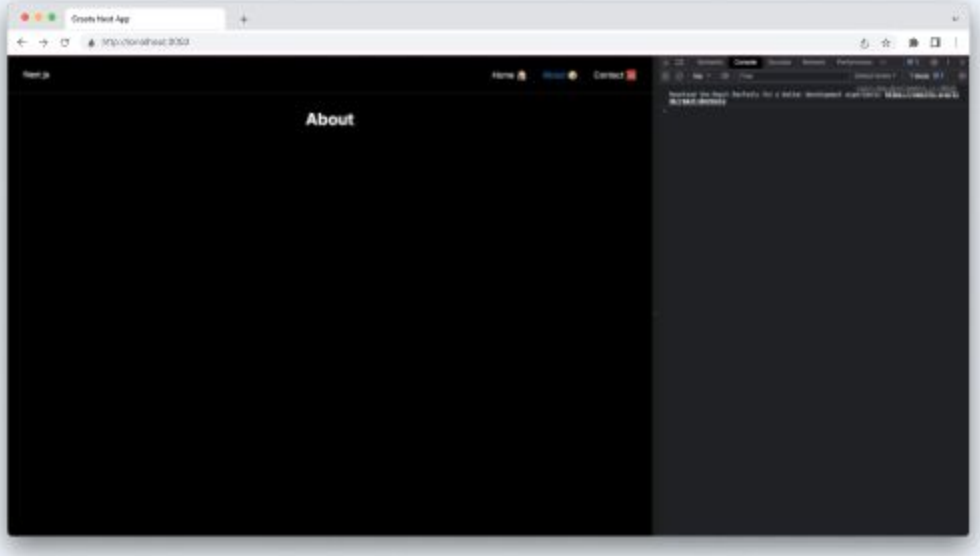
```
export default function page() {
    return (
        <main>
            <h1>Home</h1>
        </main>
    )
}
```





And what next?

Well, that's it! Go to your browser and click the "About 🐶" link in the Navbar. You will notice that the URL changes, and the text displayed on the page switch from "Home" to "About"!



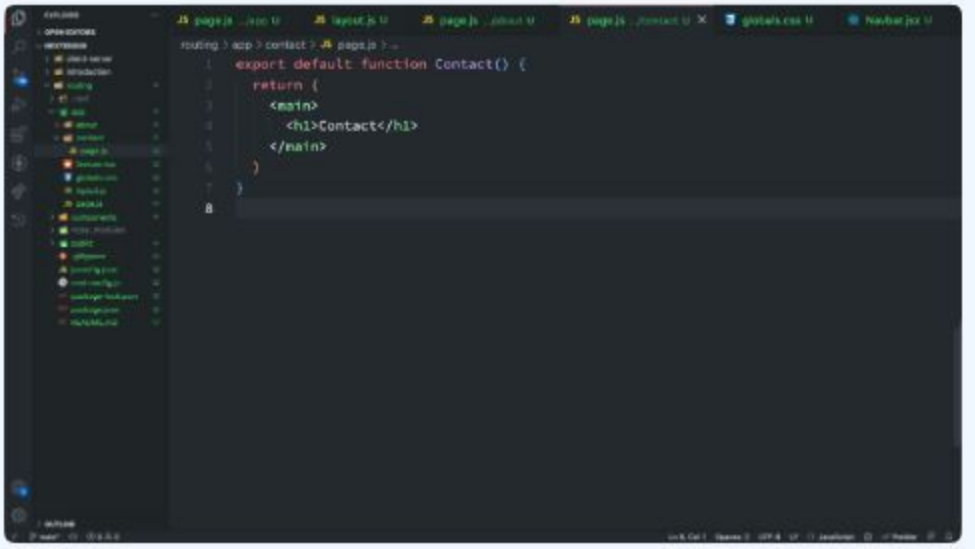
How easy is that? Especially compared to the React.js code we examined at the start!

Go ahead and create the route for "Contact 🧰" in the same way, i.e.,

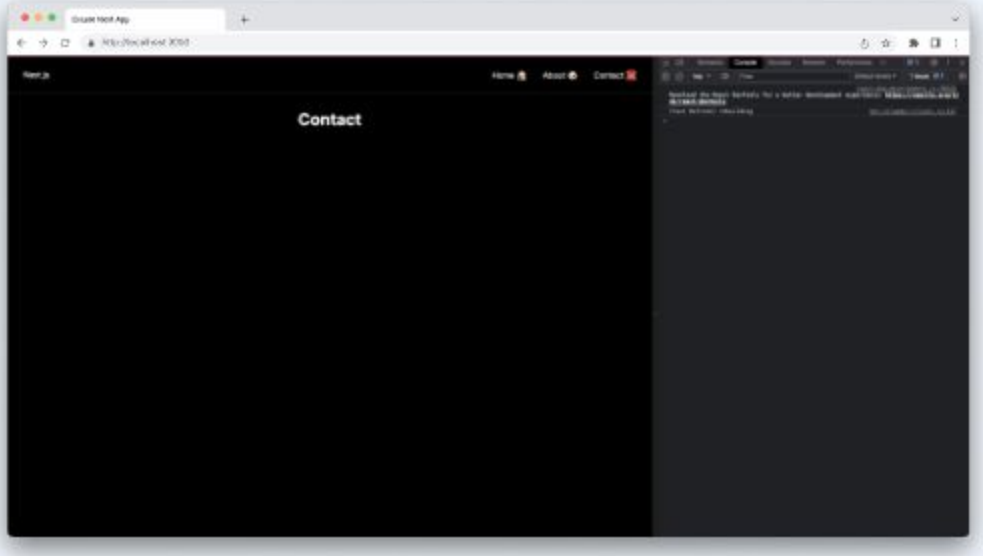
- Within the **app** directory, create a new folder called **`contact`**.
- Inside the newly created **`contact`** folder, create a file named **`page.js`**.
- Add the necessary code to the **`page.js`** file to display the desired text.

app/contact/page.js

```
export default function Contact() {
  return (
    <main>
      <h1>Contact</h1>
    </main>
  )
}
```



After saving the changes, return to the browser. You will notice that the modifications are immediately visible without reloading the page, all thanks to Next.js' Fast Refresh feature.



That was all about creating a simple route in Next.js. But what about nested or dynamic routes? What do we have to do? Let's explore

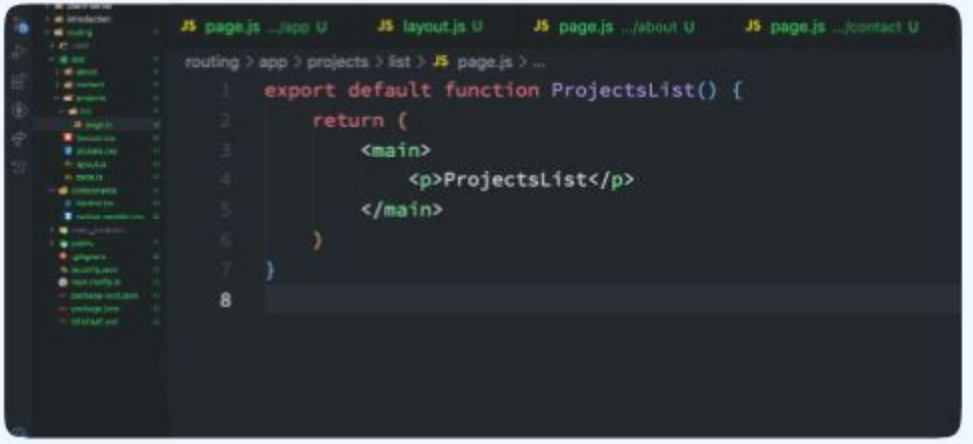
Nested Routes

It's as simple as nesting one folder inside the other.

For instance, we wish to set up a route named projects/list. To achieve this, we create two folders: projects and within it another folder called list. Inside the list folder, we add the page.js file containing the user interface (UI) for that specific route.

app/projects/list/page.js

```
export default function ProjectsList() {
    return (
        <main>
            <p>ProjectsList</p>
        </main>
    )
}
```

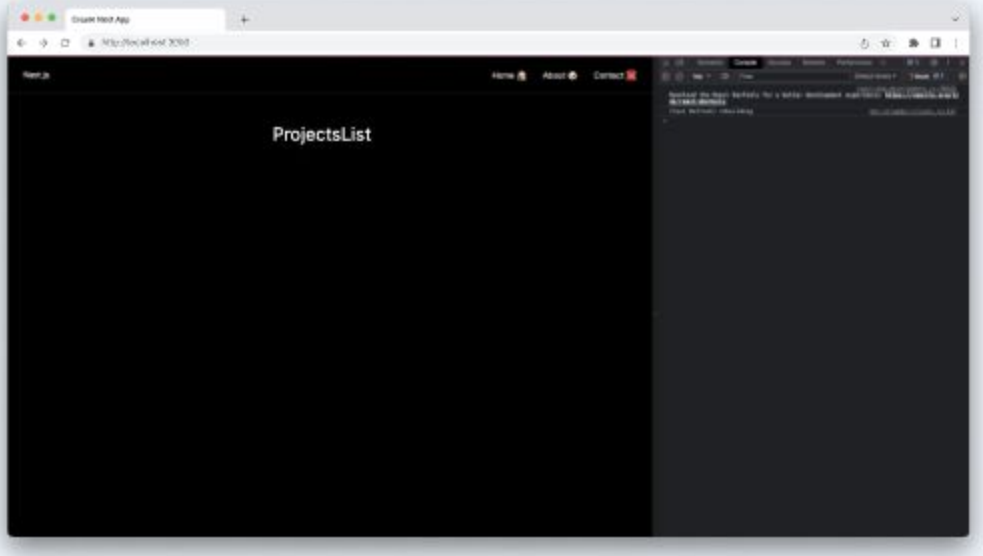


To simplify navigation to the Product List page, let's quickly include a Link tag within the Home page:

app/projects/list/page.js

```
import Link from "next/link";

export default function Home() {
  return (
    <main>
      <h1>Home</h1>

      <Link href="/projects/list">
        See Projects
      </Link>
    </main>
  )
}
```

Now visit the home page, and you'll see **See Projects**; clicking on this link will take you to the route we created, which is **/projects/list**.



Simple nest folders within one another and create whatever route you want. Moving forward, we have,

Dynamic Routes

Think of it as nested routes but with a slight difference. Unlike traditional nested routes where we need to know the exact route name in advance, dynamic routes allow for more flexibility.

The route is determined based on changing data in the application, so we don't need to predict it beforehand.

For instance, if we need to show various project details, we can design a single details page with a consistent layout for all projects. The only difference will be some data that changes for each project. Instead of making separate routes for every project detail page, we can use a dynamic route of Next.js.

To create a dynamic route, we'll have to wrap the folder's name in square brackets, symbolizing that the content inside this square bracket is variable, i.e., [folder-name].

Continuing our current application, let's add a feature for displaying project details. Imagine we have three projects named **`jobit`**, **`carrent`** and **`hipnode`**. We need three routes to showcase these projects: **`/projects/jobit`**, **`/projects/carrent`**, and **`/projects/hipnode`**. Each route represents a different project, allowing us to show its details.

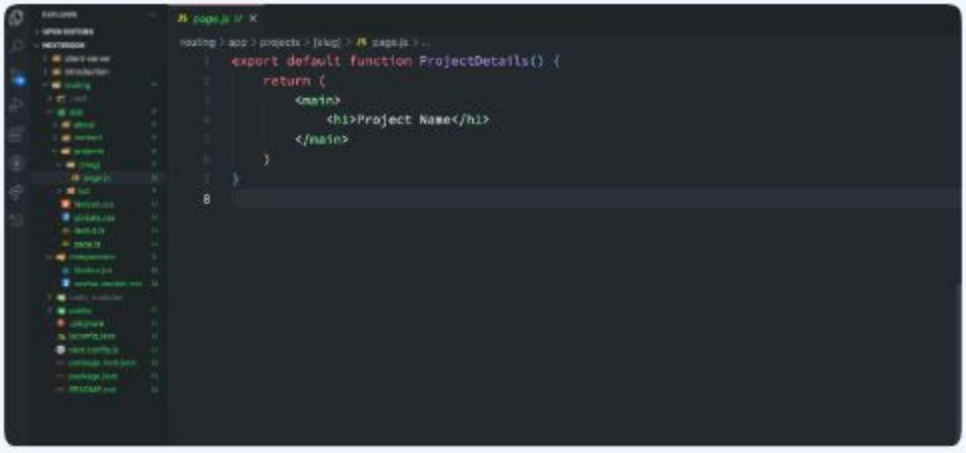
We could create these as nested routes, but there are better ways.

Imagine the number of folders you'd have to create if you're a professional developer with over 10 fantastic projects! And then, you'd have to keep copy-pasting the similar project details code. That's where dynamic routes come to the rescue! They provide a better solution for handling such scenarios.

Within the existing **`projects`** folder, create a new folder with a name enclosed in square brackets, i.e., **`app/project/[slug]`**. I referred to it as **`slug`** to address the segment in general, but you can choose any name you prefer, like **`id`** or **`name`**. Additionally, create the corresponding special UI file, **`page.js`**, in the same location folder.

app/projects/[slug]/page.js

```
export default function ProjectDetails() {
    return (
        <main>
            <h1>Project Name</h1>
        </main>
    )
}
```



To access this route, we will include links to our hypothetical projects on the projects list page and give them a bit of styling.

app/projects/list/page.js

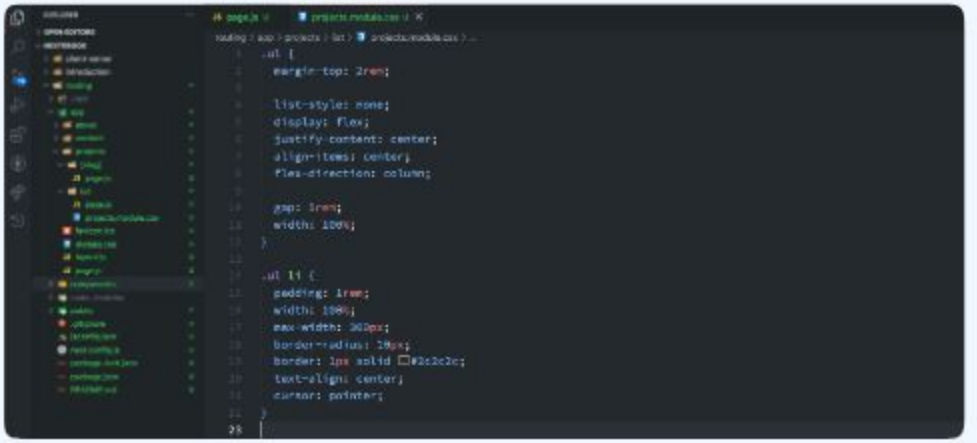
```
import Link from "next/link";
import styles from './projects.module.css'

export default function ProjectsList() {
    return (
        <main>
            <h1>My Projects</h1>

            <ul className={styles.ul}>
                <li>
                    <Link href='/projects/jobit'>
                        JobIt
                    </Link>
                </li>
                <li>
                    <Link href="/projects/carrent">
                        Car Rent
                    </Link>
                </li>
                <li>
                    <Link href="/projects/hipnode">
                        hipnode
                    </Link>
                </li>
            </ul>
        </main>
    )
}
```

And corresponding relatively simple styles in the same folder:

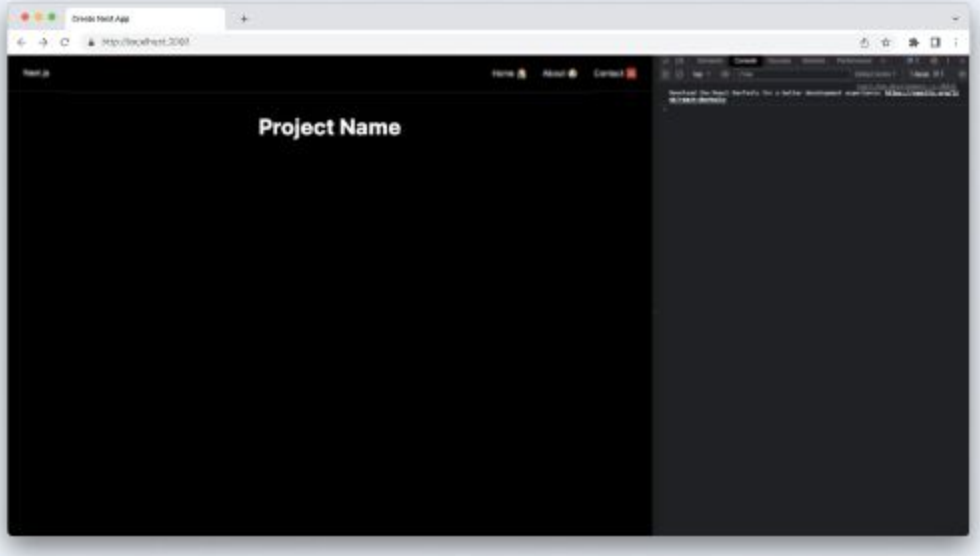
app/projects/list/projects.module.css

```css
.ul {
  margin-top: 2rem;

  list-style: none;
  display: flex;
  justify-content: center;
  align-items: center;
  flex-direction: column;

  gap: 1rem;
  width: 100%;
}

.ul li {
  padding: 1rem;
  width: 100%;
  max-width: 300px;
  border-radius: 10px;
  border: 1px solid #2c2c2c;
  text-align: center;
  cursor: pointer;
}
```

And corresponding relatively simple styles in the same folder:



Is there anything else? Nope, that's all there is to it. Simply click on these project names and witness it in action! 😃



However, something doesn't feel good. While the route for these projects changes correctly, it would be great to see the actual project name displayed on the respective route pages instead of the static "Project Name." *So, how to do that?*

The **`[slug]`** part over here is our dynamic route segment. And Next.js provides a way to access what value has been passed to it via the **`params`** prop passed to **`page.js`** page.

To utilize the value of this dynamic segment, we need to do this:

app/projects/[slug]/page.js

```
export default function ProjectDetails({ params }) {
    return (
        <main>
            <h1>Project {params.slug}</h1>
        </main>
    )
}
```

If you choose to use **[id]** or **[name]** instead of **[slug]** as the folder name, you will need to access it as **params.id** or **params.name**, respectively. Whichever name you provide, it will be the same name to access the value through the params object.

Amazing, isn't it?

But that's not the end of the routing in Next.js. Coming next are,

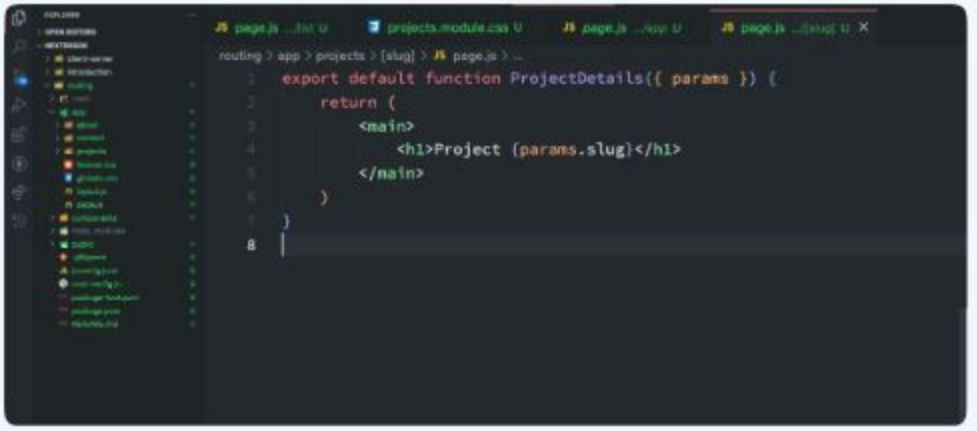
Route Groups

When working with a file-based system, having numerous folders within the app folder may be better, especially in a more complex system. To address this and offer better control over folder organization without impacting the URL path structure, Next.js introduced a feature called "Route Groups."

Need help to make sense of?

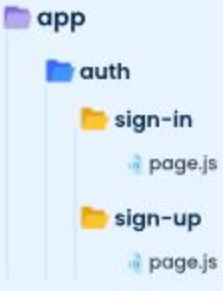
Consider the existing structure: we already have three folders: **about**, **contact**, and **projects**.

Now, if we need to add functionality like **sign-in** and **sign-up**, we would have to create more folders within the app folder, causing it to grow larger and larger.



What if we could limit the number of folders inside the app folder to a maximum of 1-3 and include everything within these folders while maintaining the same route path?

In this case, if we create these folders, i.e., **auth** and, let's say, **dashboard**, right away and add the corresponding folders & pages inside them, it will impact our routing. Why? Because, as we've learned, each folder name serves as a route name.



If we do it like the above, the route name for the sign-in page would be **`/auth/sign-in`**, and similarly, for the sign-up page, it would be **`/auth/sign-up`**.

We intended something else, right? Our desired route names are **`/sign-in`** and **`/sign-up`**, but we still want to maintain proper file organization. We don't want **`auth`** to be included in the URL, but we do want it to be present in our code structure.

To meet this specific requirement, we have **Route Groups**. They help organize routes into logical groups like **`auth`**, **`team`**, etc. We can create a route group by enclosing the folder name in parentheses, like (**`auth`**).

In this case, if we create these folders, i.e., **`auth`** and, let's say, **`dashboard`**, right away and add the corresponding folders & pages inside them, it will impact our routing. Why? Because, as we've learned, each folder name serves as a route name.

1. (auth)

Create the (**`auth`**) folder inside the **`app`** folder and add routes for the **`sign-in`** and **`sign-up`** pages. Additionally, create a **`page.js`** file within each of these folders to display the respective UI:

- Sign In
- Sign Up

- Sign In

app/(auth)/sign-in/page.js

```
export default function SignIn() {
    return (
        <main>
            <h1>Sign In</h1>
        </main>
    )
}
```

- Sign Up

app/(auth)/sign-up/page.js

```
export default function SignUp() {
    return (
        <main>
            <h1>Sign Up</h1>
        </main>
    )
}
```

Here is how the structure should appear:

Next, let's transfer the remaining folders, namely about, contact, and projects, into our (dashboard) route group.

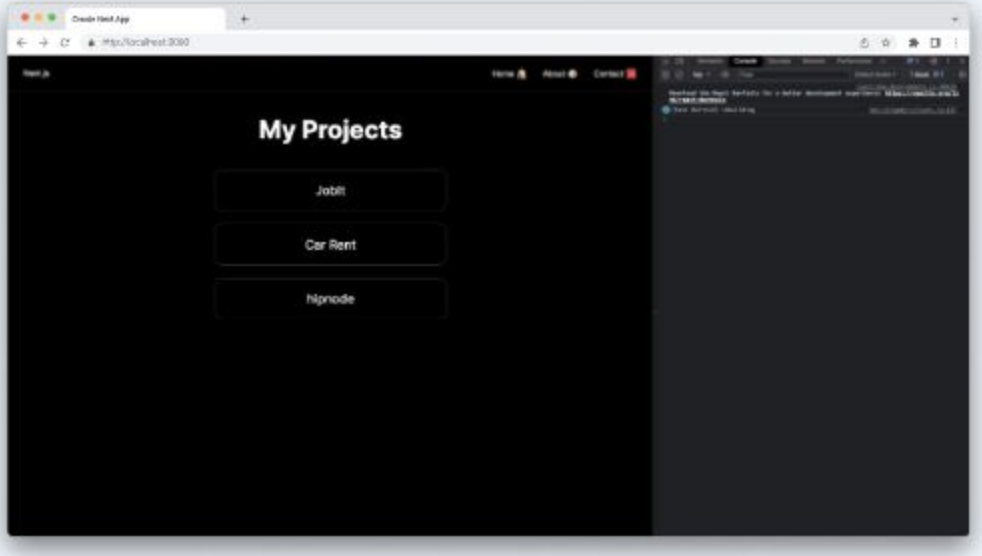
2. (dashboard)

Create a folder named (**dashboard**) within the (**app**) folder and simply move the previously created folders (**about**, **contact**, and **projects**) into it as they are.

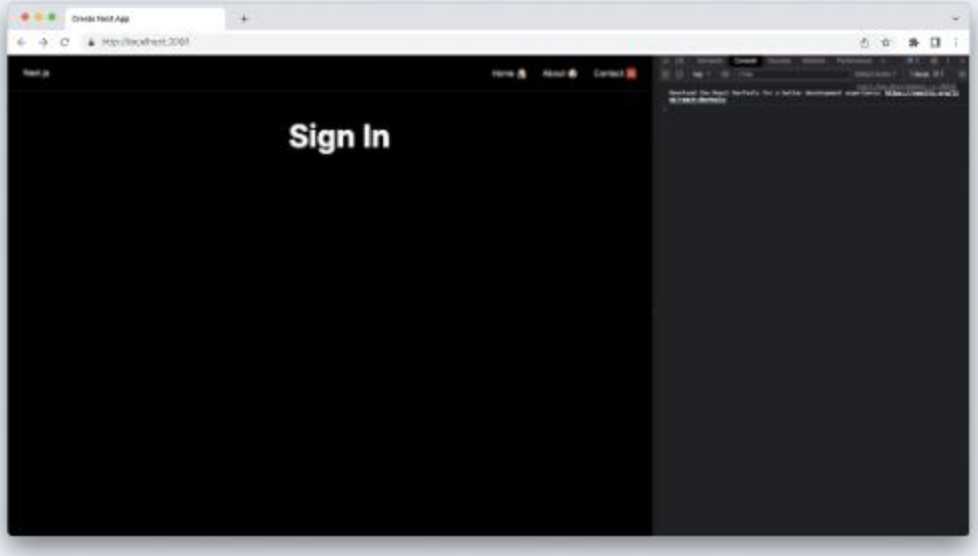
Your structure should now look like this:

All set. Now let's put our application to the test 🤞

How surreal! No code breaks, and everything works flawlessly — as if nothing had happened. The URL, the linking — everything is functioning perfectly, and all of this on the first try! 😀



Moreover, we can observe our newly created auth routes by modifying the URL, such as:



That, my friend, is the beauty of Next.js 15!

Is that the end? End of routing? **Of course, not!**

There are two more amazing client-side routing features, i.e., Parallel Routes and Intercepting Routes. Not to forget, we also have API routes 😉

We will discuss the Parallel & Intercepting Routes in the "Advanced Routing" chapter with the suitable associated code example. But to warm you up:

Parallel Routes

This feature allows us to display one or more pages simultaneously or conditionally within the same layout.

Let's imagine we're developing an e-commerce dashboard. Depending on the logged-in user, we need to render different UI components. For instance, if an admin is viewing the dashboard, we want to display complete sales data and the list of users and products. However, for non-admin users, we should show sales and products specific to them while hiding the list of users. All of this should be on the same route.

Rather than complicating the code within a single page with multiple conditions, we can utilize parallel routes that render based on whether the user is an admin or not.

Intercepting Routes

This feature is handy when displaying a new route while preserving the current page's context. It allows us to intercept a new route without fully transitioning from the current layout.

Now, let's consider a scenario where we're developing an e-commerce website and want to implement a product preview feature. When we click on the "Preview" button for a product, it should display limited information about the product in a modal format, and the URL should change to products/product-1. However, the modal should remain on top of the current page, just like a typical modal.

This is where intercepting routes come in handy.

We can utilize this feature to display the content inside the modal on top of the page from which it was triggered while also updating the URL to reflect the product previewed.

We'll fully dive into these two powerful features when creating an application in the upcoming advanced routing chapter.

Tasks 🎯

💻 Create a complete routing structure for an e-commerce project using different routes. Here are the expected routes:

- Home page: **`"/"`**
- Product listing page: **`"/products"`**
- Product detail page: **`"/products/{productId}"`**
- Shopping cart page: **`"/cart"`**
- Checkout page: **`"/checkout"`**
- Order confirmation page: **`"/order/{orderId}"`**
- User account page: **`"/account"`**
- Login page: **`"/login"`**
- Registration page: **`"/register"`**
- Search results page: **`"/search?q={searchQuery}"`**

❓ Explore routing of Next.js. How does it differ from routes in React.js?

❓ What is the purpose of route groups, and how can they be created in Next.js?

❓ What is a dynamic route, and why should we create dynamic routes in web applications?

CHAPTER 8

Rendering

In this chapter, you'll learn about rendering in Next.js and gain a deep understanding of key concepts, strategies, and environments. You'll discover how Next.js handles rendering, the different rendering strategies it offers, and when to use each one.

Rendering

We have previously discussed terms like "rendering," "runtime, and "environment," but what do they truly mean, and how does Next.js fit into the picture?

You might be thinking, "Enough with the theory, show me the code!" Well, we can definitely do that. In fact, we have already done some best-in-class Next.js 15 project videos for you to dive right into.

Build and Deploy a Full Stack MERN Next.js Threads App...

Watch and Code Now ↗

Build and Deploy a Full Stack Next.js Application | React...

Watch and Code Now ↗

Build and Deploy a Modern Next.js Application | React, Next JS, Typ...

Watch and Code Now ↗

Next.js Full Course 2024 | Build and Deploy a Full Stack App...

Watch and Code Now ↗

However, it's important to note that simply watching these videos and successfully deploying your application might not suffice.

When you eventually venture into your own projects, you might stumble because you lack a deep understanding of the "why" behind your decisions. You'll find yourself struggling with the choice.

So always aim to clear your "Why" and sit back to watch yourself perfecting the "How"!

In Next.js 15, there are different ways things are displayed (strategies), the specific times they run (runtime/build time), and the specific places where they work (environment).

Rendering

It's a process of generating or creating the user interface from the code we write. React 18 and Next.js 15 introduced different strategies to render an application. Believe it or not, we can use multiple strategies within the same application to render it differently — the god mode feature of Next.js!

Although we did talk about it a bit,

Environments

There are two environments where we can render our application code, i.e., the client (User's browser) and server (Computer where we deploy our code).

	Client	Server
Rendering Process	Occurs on the user's browser	Happens on the server before sending the page to the client's browser
Interactivity & Load Time	Provides a dynamic and interactive user experience	Provides a fully rendered HTML page to the client resulting in faster initial page load time
Fetching & SEO	Smoother transition between the pages and real-time data fetching	Fully rendered content enhancing search engine rankings and social media sharing previews
Load & Performance	Reduced server load and potentially lower hosting costs as the client's browser is responsible for handling the rendering.	Performs well on any slower device as rendering is done on the server
Consistent Rendering	Compatibility and performance depend on the user's device configuration.	Consistent rendering across any devices regardless of the configuration reducing the risk of compatibility issues
Security	Potential risk of security vulnerabilities such as Cross-Site Scripting (XSS), Code Injection, Data Exposure, etc.	Reduces the amount of client-side JavaScript code sent to user's browser thus enhancing security by limiting potential vulnerabilities

So which to use and when?

If search engine optimization (SEO), security concerns, and user device specifications are not a priority for you, and your focus is primarily on delivering dynamic interactivity to the user, then client-side rendering (CSR) with technologies like React.js can be a suitable choice.

A use case where this approach is applicable is in the business-to-business (B2B) domain. In such cases, the target audience is specific and known, eliminating the need to prioritize SEO since the product is not intended for a wide public audience. This allows you to prioritize developing interactive features and functionalities without dedicating significant resources to SEO optimization.

And if you're someone who cares about all these points, well, you know what to choose 😃

The time

Once the compilation process is complete, which involves converting code from a higher-level programming language to a lower-level representation (binary code), our application goes through two crucial phases: Build Time and Run Time.

Build time

It's a series of steps where we prepare our application code for production involving the steps of code compilation, bundling, optimization, etc.

In short, build time or compile time is the time period in which we, the developer, is compiling the code.

Remember the **`npm run dev`** script?

It's that command that generated the build of our application containing all the necessary static files, bundling, optimization, dependency resolution, etc.

Run Time

It refers to the time period when the compiled or deployed application is actively executing and running, involving the dynamic execution of the application's code and utilization of system resources.

In short, run time is the time period when a user is running our application's piece of code.

It's about handling user interaction, such as user input, responding to events, to data processing, such as manipulating/accessing data and interacting with external services or APIs.

Run Time Environment

Don't confuse this with the "Run Time" we talked about just before. That was the time period of an application. Whereas RTE, run time environment, is a specific environment in which a program or application runs during its execution.

It provides a set of libraries, services, or runtime components that support the execution of the program.

The Node.js — What is it?

It's a JavaScript Run Time Environment that allows us, developers, to run JavaScript code outside of the web browser.

Similarly, Next.js provides two different run time environments to execute our applications' code.

The Node.js runtime

Default runtime that has access to all Node.js APIs and the ecosystem

The Edge runtime

A lightweight runtime based on Web APIs with support to a limited subset of Node.js APIs.

Next.js offers the flexibility of choosing the runtime. You can do switch swiftly by changing one word:

```
export const runtime = 'edge' // 'nodejs' (default) | 'edge'
```

Isn't it amazing? Just with a simple word change, a whole new ecosystem emerges. It's like the snap of Thanos's fingers, and suddenly, a completely different world opens up!

And for the final,

Rendering Strategies

Depending on the above-discussed factors, such as the rendering environment, and the time period, i.e., build and run time, Next.js provides three strategies for rendering on the server:

Static Site Generation

Remember the build time? Well, the famous SSG, static site generation, happens at build time on the server.

During the build process, the content is generated and converted into HTML, CSS, and JavaScript files. It doesn't require server interaction during runtime. The generated static files can be hosted on content delivery network (CDN) and then served to the client as-is.

The result, the rendered content, is cached and reused on subsequent requests leading to fast content delivery and less server load. This minimal processing results in higher performance.

Although SSG handles dynamic data during the build process, it requires a rebuild if you update anything, as it happens during the build time!

An example use case would be any Documentation or Blog & News websites. All the articles or content are static 90% of the time. It doesn't need any processing. Once built, we can ship it as it is. Whenever we want to update the content, we can rebuild it!

To address this limitation, Next.js introduced,

Incremental Static Generation

It allows us to update these static pages after we build them without needing to rebuild the entire site.

The on-demand generation of ISR allows us to generate a specific page on-demand or in response to a user's request. Meaning, a certain part of the websites or pages will be rendered at build time while other is generated only when needed, i.e., at run time.

This reduces the build time and improves the overall performance of the website by updating only requested pages for regeneration.

With this hybrid strategy, we now have the flexibility to manage content updates. We can cache the static content as well as revalidate them if needed.

An example use case would be the same where we can use SSG for the article details page and use ISG for showing a list of articles

And last but not least,

Server Side Rendering

Dynamic rendering, in a nutshell, enables the generation of dynamic content for each request, providing fresh and interactive experiences.

If we have SSG and ISG, why do we need SSR?

Given the availability of Static Site Generation (SSG) and Incremental Static Generation (ISG), one might wonder why Server Side Rendering (SSR) is still needed. Both approaches offer valuable benefits, but their suitability depends on specific use cases.

SSR excels in situations where a website heavily relies on client-side interactivity and requires real-time updates. It is particularly well-suited for authentication, real-time collaborative applications such as chat platforms, editing tools, and video streaming services.

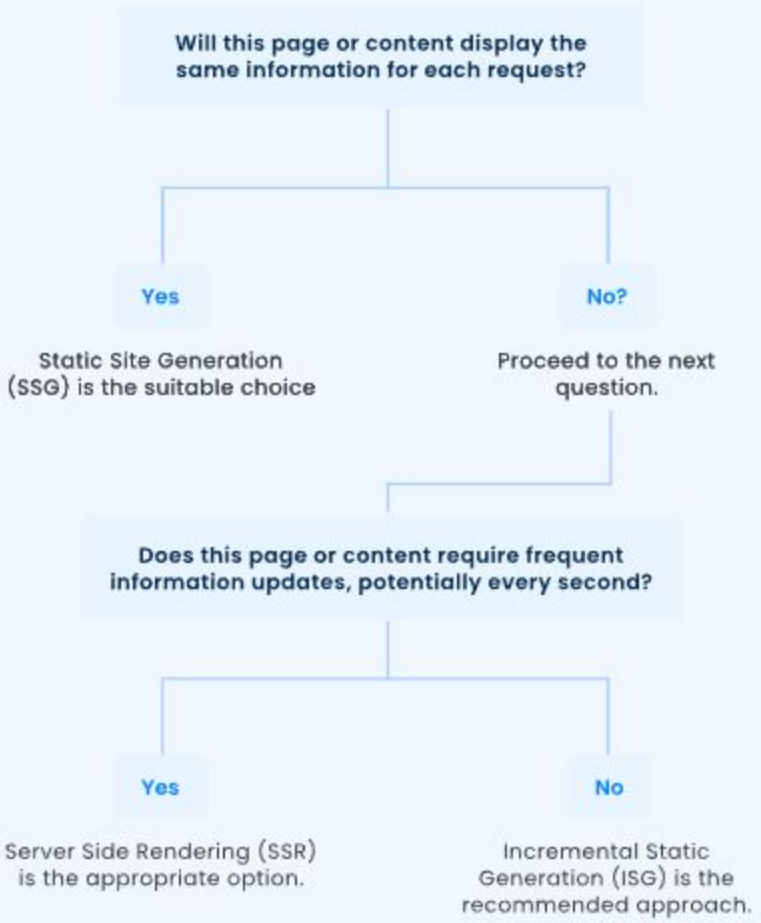
SSR involves heavy server-side processing, where the server executes code for every individual request, generates the necessary HTML, and delivers the response along with the required JavaScript code for client-side interactivity.

Due to this dynamic nature, caching content responses becomes challenging, resulting in increased server load when compared to SSG or ISG. However, the benefits of real-time interactivity and up-to-date content make SSR a valuable choice for specific application requirements.

But hey, we have the freedom to choose any of these rendering techniques for any part of your page code! Yes, you read that right. By default, Next.js uses Static Site Generation rendering.

However, we can easily switch to Incremental Static Generation or Server Side Rendering as per our specific requirements for different parts of your application. The flexibility of Next.js allows us to pick the most suitable rendering approach for each page of our website.

Okay, okay, but when to use which method?



That wraps it up, my friend. By understanding what it is and when to utilize it, we can make informed decisions that will impress our managers and bosses. 😃

The how of all this is following in the next chapter, so keep reading. But before you go, as usual, pause and try to answer these questions:

Tasks 🎯

- What does rendering mean? Explain different rendering strategies of Next.js
- What is build time and run time? Explain the difference between them in a Web application life
- What are the benefits of rendering content in a Client vs Server environment?
- Imagine, you are developing a large-scale e-commerce platform that requires a rendering strategy to handle a high volume of product listings. The platform needs to display product information, pricing, availability, and customer reviews. Additionally, the platform aims to provide a fast and interactive user experience.

  Considering the complex requirements of the e-commerce platform, discuss the trade-offs and factors you would consider when choosing between Static Site Generation (SSG) and Server Side Rendering (SSR) as the primary rendering strategy.

CHAPTER 9

Data Fetching

This chapter explores data fetching strategies in Next.js. We'll compare React's traditional hooks with Next.js' React Server Components (RSC) for efficient API and database interactions. Learn about Next.js' caching methods and rendering strategies like SSG, SSR, and ISR to enhance performance and data reliability in your applications.

Data Fetching

Earlier, we explored various rendering environments and strategies, enabling us to determine the optimal approach for maximizing our app's performance in different scenarios.

But how do we actually use these strategies in code? It's easy. The hard part was understanding how these different strategies work.

So, let's begin.

Typically, if someone asks you to implement data fetching in an app, this is likely what you'd do, right?

App.js

```
import React, { useState, useEffect } from 'react';

function App() {
  const [data, setData] = useState(null);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('https://api.example.com/data');
        const jsonData = await response.json();
        setData(jsonData);
      } catch (error) {
        setError(error);
      } finally {
        setIsLoading(false);
      }
    };

    fetchData();
  }, []);
```

```
  if (isLoading) {
    return <div>Loading...</div>;
  }

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

  return (
    <div>
      <h1>Data from API:</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}

export default App;
```

Yep, it's the usual fetch using **`useEffect`**, where we call the API when the component mounts, retrieve the data, store it in the state, and then display it.

But the same thing, with RSC — React Server Components in Next.js would look like this,

```
import React from 'react';

async function App() {
  const response = await fetch('https://api.example.com/data');
  const jsonData = await response.json();

  return (
    <div>
      <h1>Data from API:</h1>
      <pre>{JSON.stringify(jsonData, null, 2)}</pre>
    </div>
  );
}

export default App;
```

Isn't it simple? Less code? And straight to the point?

With React Server Components, we directly do fetch or any operations that we want to do without any need for hooks.

For example, if we want to make a call to our database and get a list of items, we can do this in RSC,

```
import React from 'react';
import prisma from './prisma';

async function App() {
  const users = await prisma.user.findMany();

  return (
    <div>
      <h1>Data from Prisma:</h1>
      <pre>{JSON.stringify(users, null, 2)}</pre>
    </div>
  );
}

export default App;
```

We know JavaScript follows sequential order. It starts from the top to bottom and executes each line depending on whether it's synchronous or asynchronous in behavior.

Here's a general overview of how JavaScript execution works:

1. **Parsing**: The JavaScript engine parses the entire script to understand its syntax. This involves breaking down the code into tokens and creating an Abstract Syntax Tree (AST).

2. **Execution Context Creation**: Before executing any code, the engine creates a global execution context. This context includes the global object (`window` in browsers, `global` in Node.js), the `this` keyword (which refers to the global object in the global context), and a reference to the outer environment.

3. **Hoisting**: Variable and function declarations are hoisted to the top of their respective scopes. This means that they are processed before the execution of the code starts.

4. **Execution Phase**: The code is executed line by line, starting from the top of the script. Each statement is executed in order unless it involves asynchronous operations.

5. **Asynchronous Operations**: When encountering asynchronous operations like timers (`setTimeout`, `setInterval`), AJAX requests, or event listeners, JavaScript doesn't wait for them to complete. Instead, it continues executing the rest of the code. Once the asynchronous operation is complete, a callback function is added to the event queue.

6. **Event Loop and Callback Queue**: JavaScript runtime maintains an event loop that continuously checks the call stack and the callback queue. If the call stack is empty, it takes the first callback from the queue and pushes it onto the call stack for execution.

7. **Function Execution**: When a function is called, a new execution context is created for that function, which follows the same steps as the global execution context.

Function execution contexts are pushed onto the call stack, and when a function returns, its context is popped off the stack.

8. **Scope Chain**: During execution, JavaScript resolves variable and function references by traversing the scope chain, which includes the local scope, the outer (enclosing) function scopes, and finally the global scope.
9. **Garbage Collection**: As the code executes, the JavaScript engine keeps track of objects and variables no longer in use. These are marked for garbage collection to free up memory.

For the above code, JavaScript will do the same (of course, Next.js & React.js will act first to execute code to turn this component into vanilla JavaScript), but where? Any guess?

Of course, you know. **On Server!!**

By default, any kind of components you'll write in Next.js are React Server Components which are guaranteed to run only on Server.

Then how can we write components that run on a client in Next.js? 🤔

As easy, okay, I really need to stop doing that.

We do so by adding a flag at the top of that component file, like this,

App.js

```
'use client';

import React, { useState, useEffect } from 'react';

function App() {
  const [data, setData] = useState(null);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('https://api.example.com/data');
        const jsonData = await response.json();
        setData(jsonData);
      } catch (error) {
        setError(error);
      } finally {
        setIsLoading(false);
      }
    };

    fetchData();
  }, []);

  if (isLoading) {
    return <div>Loading...</div>;
  }

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

  return (
    <div>
      <h1>Data from API:</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}

export default App;
```

Isn't it too good? We can change the rendering execution environment just by adding/removing a two-word flag for each component in Next.js.

I know, I know. But are you aware that what we've essentially done above is implement one of the rendering strategies of Next.js?

Absolutely, it's SSG (Static Site Generation), the default rendering strategy used by Next.js.

As we've previously covered, this strategy involves rendering the component on the server side during build time. Subsequently, whenever a user requests this specific component or page, the pre-generated static content is served to the user. It's cached and doesn't undergo re-rendering with each request, distinguishing it from other approaches.

Alright, I see you understand now. But what if we need to refresh or reprocess the component or content every time a user sends a request? Yes, how do we implement the SSR strategy?

Server Side Rendering

Simple. Yes, I know I said it again, but it really is as simple as flipping a light switch.

There are two ways to do this:

1. On Demand

We can explicitly tell Next.js not to cache anything and thus render content on each request to get fresh data

```
import React from 'react';

async function App() {
  const response = await fetch('https://api.example.com/data', {
    cache: 'no-store',
  });
  const jsonData = await response.json();

  return (
    <div>
      <h1>Data from API:</h1>
      <pre>{JSON.stringify(jsonData, null, 2)}</pre>
    </div>
  );
}

export default App;
```

cache has two values,

- **`no-store`**: With "no-store," Next.js skips checking its cache memory altogether. Instead, it goes straight to the main server every single time you ask for something. Plus, it doesn't save anything it downloads, so there's no chance of using cached data later on.
- **`force-cache`**: When setting the cache value to this, Next.js checks if it already has the information we're asking for stored in its memory. If it does and it's still up-to-date, it'll just give us that info without bothering anyone else. But if it doesn't have what we need, or if what it has is outdated, it'll go out and get the latest version from the main server, then save that updated info as a cache for next time.

There are additional methods, such as **`revalidatePath`** or **`revalidateTag`**, for on-demand validation, but we'll dive into those later in the "Server Actions" chapter.

2. Time based

The whole basis of these rendering strategies is to cache the data depending on time.

We know that by default, Next.js uses SSG, which caches everything. So what if we define this method at the time we want? Means, revalidate the cache whenever we want to.

And how do we do that? As simple as... Yes, I know, another analogy. But it's really that simple.

We do it by adding this statement on the page,

```
export const revalidate = false | 0 | number;
```

or if we're using fetch API, we do it so by,

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

Do keep in mind that the fetch we're using here isn't the actual Web API Fetch. It has been extended by Next.js to automatically cache requests. It's an extension to the original fetch API.

So if you see someone using fetch in Next.js applications instead of axios or other, then it's there for some reason.

Did you read the above callout? Don't skip it 😄

Back to the revalidate, it accepts different values,

- **false:** Means, as you expected, the resource will be stored in the cache for as long as possible, basically forever. It's like saying "keep this forever" to Next.js.

  But "forever" may not always work (like the real world). Although Next.js will keep the cache, HTTP cache might decide to get rid of older stuff eventually to make room for new things.

  The HTTP cache is a mechanism used by web browsers and servers to store previously accessed web resources, such as HTML pages, images, CSS files, and JavaScript files.

  When we visit a website, our browser may store copies of these resources locally on our device to speed up future visits. This caching helps reduce load times and bandwidth usage by retrieving files from the local cache rather than downloading them again from the web server.

  One more thing, **`revalidate=false`** is the same as **`revalidate=Infinity`**. Different ways of achieving the same thing.

- **0:** It's similar to setting **`revalidate`** to false. Same effect. It'll tell Next.js not to cache anything and render that specific page/route every time the user makes a request/visit
- **number:** Choosing a certain number tells Next.js how long to keep a page or route in its memory (cache) before refreshing it. For example, if we set it to 5 minutes, Next.js will remember and show the same page for 5 minutes. After that, it will get a fresh version, save it, and repeat the loop. Understand? Does that remind you of something?? Think!!

Now moving to where we were actually, i.e., Server Side Rendering. So can you now give me an example of how we can do SSR with time-based validation?

Yes, we can implement SSR like this,

```
import React from 'react';

async function App() {
  const response = await fetch('https://api.example.com/data', {
    next: { revalidate: 0 },
  });
  const jsonData = await response.json();

  return (
    <div>
      <h1>Data from API:</h1>
      <pre>{JSON.stringify(jsonData, null, 2)}</pre>
    </div>
  );
}

export default App;
```

And yes, that way too,

```
import React from 'react';

export const revalidate = 0;

async function App() {
  const response = await fetch('https://api.example.com/data');
  const jsonData = await response.json();

  return (
    <div>
      <h1>Data from API:</h1>
      <pre>{JSON.stringify(jsonData, null, 2)}</pre>
    </div>
  );
}

export default App;
```

Amazing, both answers are absolutely correct to render a page every time a user makes a request!

Upcoming,

Incremental Static Regeneration

Do you remember what and how ISR works?? We actually learned about it in SSR too. Take a pause, answer it to yourself, and then keep reading.

When we want to make the most of both SSG and SSR, we do ISR. For example, for blogging applications. And a way to implement that in Next.js is like this,

```
import React from 'react';

export const revalidate = 5000; // (seconds)

async function App() {
  const response = await fetch('https://api.example.com/data');
  const jsonData = await response.json();

  return (
    <div>
      <h1>Data from API:</h1>
      <pre>{JSON.stringify(jsonData, null, 2)}</pre>
    </div>
  );
}

export default App;
```

And if we just want to implement ISR only for a particular fetch, we do,

```
import React from 'react';

async function App() {
  const response = await fetch('https://api.example.com/data', {
    next: { revalidate: 5000 },
  });
  const jsonData = await response.json();

  return (
    <div>
      <h1>Data from API:</h1>
      <pre>{JSON.stringify(jsonData, null, 2)}</pre>
    </div>
  );
}

export default App;
```

Easy peasy, isn't it??

Time to revise what you learned in this chapter. Take a pause and answer these questions.

Tasks 🎯

- What are the methods for fetching data in Next.js?
- Explain the implementation of SSG, ISR, and SSR in Next.js with examples for each.
- What is the process for data revalidation in Next.js?
- Define the distinction between Time-Based and On-Demand Validation
- Explain the difference between these two examples and when we should use which.
  - Scenario 1

```
import React from 'react';

async function App() {
  const data = await fetch("https://api.example.com/data",
   { next: { revalidate: 5000 } }
 );
  const dataJSON = await data.json();

  const users = await fetch(https://api.example.com/users")
  const usersJSON = await users.json()

  return (
    <div>
      <h1>Data from API:</h1>
      <pre>{JSON.stringify(dataJSON, null, 2)}</pre>

      <h1>Users from API:</h1>
      <pre>{JSON.stringify(usersJSON, null, 2)}</pre>
    </div>
  );
}

export default App;
```

- Scenario 2

```
import React from "react";

export const revalidate = 5000

async function App() {
  const data = await fetch("https://api.example.com/data");
  const dataJSON = await data.json();

  const users = await fetch(https://api.example.com/users")
  const usersJSON = await users.json()

  return (
    <div>
      <h1>Data from API:</h1>
      <pre>{JSON.stringify(dataJSON, null, 2)}</pre>

      <h1>Users from API:</h1>
      <pre>{JSON.stringify(usersJSON, null, 2)}</pre>
    </div>
  );
}

export default App;
```

CHAPTER 10

SEO and Metadata

This chapter explores SEO (Search Engine Optimization) in Next.js, focusing on enhancing website visibility and attracting organic traffic. Learn about implementing metadata for better search engine indexing, distinguishing between static and dynamic metadata, and leveraging Next.js features for optimized SEO strategies.

SEO and Metadata

We talked about lots of things so far, from how Next.js works to different strategies, their implementation, and all. One of the highlighted parts of Next.js we talked about how it's useful when it comes to "SEO"

Using all these features we learned is amazing, but if we people don't see it, then there is no use for it, is it?

We build websites for people. So it's also our responsibility to find a solution in such a way that we'll get lots of traffic on our website and it's noticeable by people on a seaful of websites internet.

There are many ways to do that via the "non-developer" way. Meaning, your company or your team can create ads, funny memes, or create lots of social content, stay active, and all those. But this is non-organic traffic, meaning you have to pay some to get some traffic.

On the other hand, organic traffic is what we as developers can do inside our code so web crawlers would be able to index them properly on search engines aka SEO — Search Engine Optimization.

SEO is like trying to impress a picky cat with your website. Imagine your cat is internet, and it only pays attention to the shiniest, most interesting toys (websites). So, SEO is your way of making your website the coolest, sparkliest cat toy in the room—using the right tricks (keywords), arranging things neatly (site structure), and making sure your cat enjoys playing with it (user experience).

The better you do this, the more your cat (search engine) will show off your awesome toy to others (users).

We can improve SEO by following these best practices on our website,

Best Practices to Improve SEO

Keywords

Keywords are like labels that tell web crawlers what your content is about. When a user searches for something using these words, the crawlers know to present your page as a potential match.

Content Quality

High-quality content is like a magnet for both users and crawlers. Users stick around longer, and crawlers recognize that engagement, interpreting it as a signal that your content is relevant.

Meta Tags

Think of meta tags as your page's ID card. It tells search engines and users a quick summary of what to expect. A convincing ID card means more clicks.

Website Structure

Imagine your website is a book. A well-organized structure is like chapters and headings, making it easy for search engines to read and understand what's inside.

Site Speed

Faster websites are like express lanes. Search engines love it because they can quickly check and list your pages. Users love it because they don't have to wait.

And that's what we learned so far throughout this ebook on how using RSC and SSR combined in Next.js gives us the best website performance.

Backlinks

Backlinks are recommendations. When other reputable sites link to yours, it's like saying, "This site is trustworthy." Search engines notice and boost your ranking.

Clear URLs

URLs are like street addresses. Clear ones make it easy for search engines to find your pages. Confusing addresses may lead to lost visitors.

... and a few more. But now let's see how we can actually implement some of these easily in Next.js.

There are two ways through which we can add metadata to our website using Next.js's Metadata API that decides the look & feel of our website when sharing with others or on the Internet in general (for search engines)

But before that, we need to address the difference between static and dynamic metadata. Do you know the difference between them?

Static Metadata

Simply put, it's information that doesn't change very often, if at all.

This includes things like the page title, meta description, and meta keywords. Once set, these elements remain the same unless intentionally updated by a website owner or developer.

It's like writing your name on a label that's stuck on a box - it's there to stay unless you physically change it.

Dynamic Metadata

As you can guess, it's information that can change based on various factors like user interactions, search queries, or other conditions.

This type of metadata is often generated automatically by a website's content management system (CMS) or scripting languages.

For instance, the meta description might change depending on the specific search term a user uses or based on the content of the page.

It's like a label that updates itself depending on what's inside the box or who's looking at it.

Understood? Now in Next.js, we can implement both of these using two different methods,

1. Config Based

All we have to do is create a javascript object in any layout or page file and export it. Next.js will automatically detect it and turn it into relevant meta tags for those routes

layout.tsx | page.tsx

```
export const metadata = {
  title: 'JS Mastery',
  description:
    "Gain Real-world Experience And Land That Dev Job You've Always Imagined",
};

export default function Page() {}
```

The **title** key within this metadata object will be transformed into the HTML **<title />** tag, while the description will be converted into a **<meta />** tag within the <head /> section of the HTML document.

The result of the above code will be as follows,

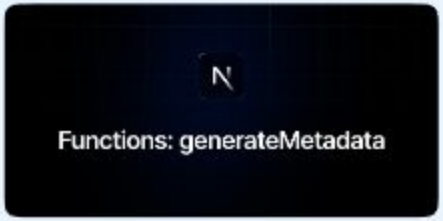
Output

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>JS Mastery</title>
    <meta name="description" content="Gain Real-world Experience And Land
     That Dev Job You've Always Imagined">
</head>
<body>
    <!-- Your page content goes here -->
</body>
</html>
```

Next.js gives us many ways to make our website better for search engines. It covers everything from setting up metadata for social media to organizing bookmarks and categories. You can check the complete metadata fields list here:

Metadata Object and generateMetadata Options

Link to blog ↗



This covered setting up static metadata using config-based approach, but how can we apply the same configuration approach to generate dynamic metadata, and when is it most appropriate to do so?

Dynamic Metadata

If there are dynamically generated pages, i.e., dynamic routes, that we want search engines to index—which is a best practice—we can use the **`generateMetadata`** function in Next.js to produce metadata specifically tailored for these pages.

For example, to show you, I have implemented this strategy on my JS Mastery website for Resources.

Free Resources | JS Mastery

Free resources database containing ebooks, guides and videos

https://www.jsmastery.pro/resources

For the main Resource page, you'll see different titles and descriptions and if now you click on one of the resources, you'll see a change in title and description as well.

Same for other resources. Each has its own title, description, and image, which helps web crawlers index these pages as well.

Okay, but how can we achieve this in Next.js? Achieving this is super easy, and it's like this:

layout.tsx | page.tsx

```tsx
export async function generateMetadata({ params }) {
  const { id } = params;
  const resource = await getResourceById({ id });

  const title = resource.title + ' | JS Mastery';
  const seoDescription =
    'Free resources database containing ebooks, guides and videos';

  return {
    title,
    description: seoDescription,
    other: {
      'og:title': title,
      'og:description': seoDescription,
      'og:image': resource.image,
      'twitter:title': title,
      'twitter:image': resource.image,
      'twitter:description': seoDescription,
    },
  };
}

export default function Page({ params, searchParams }) {}
```

The **`og:`** and **`twitter:`** fields in metadata are used for social media optimization (SMO). They provide information to social media platforms like Facebook and Twitter about how shared links from your website should be displayed.

- **`og:`**: These are Open Graph meta tags, which are used by Facebook and other platforms to understand and display shared content. Examples include **`og:title`**, **`og:description`**, and **`og:image`**. These tags help control how your content appears when shared on social media platforms.

- **`twitter:`**: These meta tags are specifically for Twitter cards, which allow you to control how your content is displayed when shared on Twitter. Examples include **`twitter:title`**, **`twitter:description`**, and **`twitter:image`**.

Prior to defining the Page component, we gather metadata by fetching data from a database, API, or any other source. We then use this data to create the metadata object, and that's all.

Some of you who have taken our Next.js course may have a question:

What happens if we make a fetch or database call in the generateMetadata function and another one in the main component to display the result? Wouldn't this result in two fetch or database calls for the same thing?

Excellent question, but the short answer is No, Nada, Nah.

As we covered in the previous lesson, fetch requests are automatically memoized by default. This isn't the standard Web Fetch; it's an extended version that caches requests across the application. So, these fetch requests are automatically memoized for the same data across **`generateMetadata`**, **`generateStaticParams`**, Layouts, Pages, and Server Components.

But, **What if we don't use fetch?**

We have alternative methods available, such as **React's Cache** or **Next.js's unstable_cache**, to achieve a similar result as Next.js's fetch and avoid calling the database or API twice on the same page.

Does this make sense?

Before we learn about the next strategy to add metadata in Next.js, do keep in mind that, "Both static and dynamic metadata through **`generateMetadata`** are only supported in Server Components."

Don't try that in Client Component and expect it to work :D

So, the next way of handling metadata in Next.js is —

2. File Based Metadata

As the name suggests, we can put files like robot, sitemap, favicon, open graph images, or other site icons directly inside the **app** folder, and next.js will automatically detect and generate the corresponding meta tags.

For example,

```
> app
  • favicon.ico
  • icon.(ico|jpg|jpeg|png|svg)
  • apple-icon.(jpg|jpeg|png)
  • opengraph-image.(jpg|jpeg|png|gif)
  • twitter-image.(jpg|jpeg|png|gif)
  • opengraph-image.alt.txt
  • twitter-image.alt.txt
  • ...
```

It's just about adding files with these names (has to be the same name) directly inside the **app** folder. It'll work out the same as the config-based approach. You can find the full list of all files that can be created to define metadata here:

File Conventions: Metadata Files | Next.js

Link to blog ↗

One thing worth noting here is, file based metadata has the higher priority and will override any config-based metadata. So if you set metadata in a file, it will be used instead of the same metadata field you set in the configuration.

So, can you now create proper SEOized applications using Nextjs? Before we dive into another interesting topic, take your time to complete these,

Tasks 🎯

- 📖 Explain the concept of search engine optimization (SEO) and its importance in web development.
- ❓ How can we implement SEO in Next.js applications?
- ❓ How does metadata impact search engine rankings and visibility of web pages?

- What are different strategies to define metadata in Next.js?
- Imagine you're hired by an online retailer that uses Next.js for their website. The company wants to improve its product pages' visibility in search engine results to drive more organic traffic and increase sales. How would you approach optimizing the product pages for SEO, considering factors such as product descriptions, images, and reviews?

CHAPTER 11

Backend

In this chapter, we dive into backend development using Next.js, exploring its modern approach to server-side operations and API handling. You'll discover how Next.js simplifies backend tasks traditionally managed with frameworks like Express.js.

Learn about setting up APIs with minimal configuration, implementing middleware for request processing, and leveraging Next.js features like dynamic routes and caching.

Backend

Having reached this point, perhaps some of you may have thought, 'Ah, I've had my fill of Frontend!'

Using all these features we learned is amazing, but if we people don't see it, then there is no use for it, is it?

I don't need to say how important it is to know about developing websites that look good if you can make **awesome sites like these** using the newest Next.js, even if you're half asleep. Then even Devin can do better than you!

Let's get back to what's important. Frontend is necessary, but so is the backend. Neither is better than the other (although I personally prefer frontend). They work together closely.

In this chapter, we'll learn how we can do "back" stuff in the latest Next.js.

We noticed a lot of changes in how things work on the front of our websites because of the new Next.js. The same kind of big changes have also happened for the behind-the-scenes part of Next.js. With new and better features, along with standard ways of doing things, the backend development has seen some really significant changes because of Next.js.

If you've ever tried some backend work, even creating a simple "Hello World from Server" message requires a fair bit of setup. It involves,

- Setting up a project
- Installing necessary packages like Express (for Node.js)
- Writing server code

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.json({ message: 'Hello, World!' });
});

app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

- Running it
- And then deploying it using some free/paid services so we could use it on frontend

It might not seem like a big deal when you're just reading or looking at the above code (for first time), but things start getting complicated when you have to write various routes, middleware, and so on.

In the newest version of Next.js, it's much like what we did on the frontend — just create a special file within a folder for the specific route, and you're all set. No need to set up, manage, or monitor an active server separately.

If we want to write the same code mentioned above in Next.js, we simply need to create a folder with any name we like and then create a special file named **`route.js/ts`** inside it. From there, we can immediately begin writing the 'server' code.

app/hello-world/route.js

```
export async function GET() {
  return Response.json({ message: 'Hello, World!' })
}
```

That's all there is to it. Your folder name serves as your API route name, with your business logic neatly encapsulated within this special route file.

This might not seem like a big deal if you've never built an API before. In our masterclass, we often share worksheets demonstrating various methods for creating backends and explaining the differences between them. This is to help people understand why we choose Next.js over Express or when Express might be the better option.

That wasn't meant to be a promotional pitch to encourage you to join us (though I'd be delighted if you did). Instead, it's to share one of our workshop Github repositories with you, so you can better understand how backend development varies across different technologies. Feel free to check it out here and experiment with it.

Worksheet Backend
Link to worksheet ↗

Moving ahead,

Nested Routes

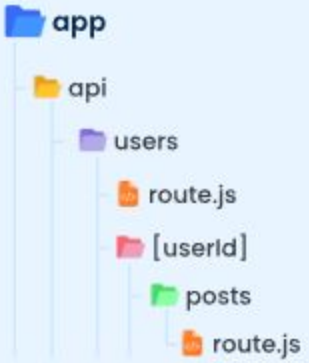
As you're aware, your folder name essentially represents a route name. If you want a route with multiple names (such as a long one), which is known as a nested route, you can create additional nested folders.

Example,

This will result in the `/api/users` API URL.

Dynamic Routes

Expanding on the nested routes mentioned earlier, and similar to frontend dynamic routes, we can create dynamic routes like this,



This will lead to the API **`/api/users/1243/posts`**. The **`[userId]`** part holds a dynamic value, any string value, depending on your **`userId`**.

And how do we access it?

By accessing the request parameters, as we typically do in Express.js using **`req.params`**.

```
export async function GET(request, { params }) {
  const userId = params.userId;
}
```

❓ Keep in mind that whatever you put inside `[]` when defining a dynamic segment will be the same name you use to access its value from the URL.

For example, if you write **`[userId]`**, you can access it using **`params.userId`**. If it's **`[slug]`**, you can access it with **`params.slug`**. It works the same for other scenarios. Any name will do!

When discussing dynamic route segments, there's another concept that often gets confused with it, and that's none other than URL search/query parameters

🧠 **Understanding API Parameters**

1. Query Parameters

- **Location**: Situated after the **?** in the URL.
- **Usage**: Used for optional data like filtering, sorting, or pagination.
- **Flexibility**: They can be added or removed without impacting the endpoint's core functionality.

- Example: **`/api/products?category=electronics`**

2. Route Parameters

- **Location**: Embedded directly within the URL path itself.
- **Usage**: Essential for specifying the resource being accessed.
- **Structure**: Integral part of the endpoint's structure, typically required.
- Example: **`/api/users/{userId}`**

When discussing dynamic route segments, there's another concept that often gets confused with it, and that's none other than URL search/query parameters

URL Query Parameters

We define URL query parameters in Next.js by using the router or the Link component to include specific query parameters.

1. Using Next.js Router

We define URL query parameters in Next.js by using the router or the Link component to include specific query parameters.

```
router.push({
  pathname: '/products',
  query: { category: 'electronics', page: 1 },
});
```

2. Using Next.js Link

```
<Link
  href={{ pathname: '/products', query: { category: 'electronics', page: 1 } }}
> Show Electronics Products </Link>;
```

3. String Interpolation

Both of the above can then be used with String Interpolation to achieve similar results of adding Query parameters to the URL

```
const category = 'electronics';
const page = 1;

router.push(`/products?category=${category}&page=${page}`);
<Link href={`/products?category=${category}&page=${page}`}>
  Show Electronics Products
</Link>;
```

That's about how we can include query parameters, which occur on the frontend side of the application.

But, **How do we access them on the backend side, through API routes?**

Short Answer — Through Request 😊

```
export function GET(request) {
  const searchParams = request.nextUrl.searchParams;
  const query = searchParams.get('query');
}
```

This isn't a new feature that Next.js introduces for backend development. In Express, for instance, we access these values through **`req.query`**. It's essentially the same process, just with a different method of retrieval.

The next interesting bit is,

Middleware

In Next.js, middleware works similarly to how it does in other frameworks (Express.js), though it's a bit different because Next.js is both a server-side and client-side rendering framework for React.

Middleware in Next.js sits between the incoming request and the route handler. It's like a series of checkpoints that the request has to pass through before reaching its final destination (the route handler or the page).

Here's a simple breakdown of how middleware works in Next.js,

1. **Request Comes In**: When a user visits a page in your Next.js application, a request is sent to the server.
2. **Middleware Execution**: Before the request reaches the actual route handler or page, it passes through one or more middleware functions.
3. **Middleware Operations**: Each middleware function can perform operations on the request, like logging, authentication, modifying request headers, etc.
4. **Passing Control or Short-Circuiting**: Depending on what the middleware does, it can either pass control to the next middleware in line or short-circuit the process and send a response back to the client without reaching the route handler.
5. **Finally Reaching Route Handler/Page**: If the request successfully passes through all the middleware functions, it finally reaches the route handler or page component where the appropriate response is generated.

Right now, while writing this ebook, we're limited to one middleware per project. And the way we can create it is by creating a **`middleware.js/ts`** file at the root of the project.

middleware.js/ts

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

export function middleware(request) {
  console.log('do your stuff')
}
```

Even though we're limited to one main middleware file, we can still organize your middleware logic into separate files. This means we can break down our middleware functionalities into smaller, more manageable pieces, each in its own file.

For example,

- **Auth Middleware**

lib/authMiddleware.js

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

export function authenticate(request) {
    console.log('Authentication middleware')
}
```

- **Logging Middleware**

lib/logMiddleware.js

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

export function logRequest(request) {
    console.log('Logging middleware')
}
```

And then our main middleware will be,

middleware.js

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

import { logRequest } from './logMiddleware'
import { authenticate } from './authMiddleware'

export default function middleware(request) {
    authenticate(request, () => {
        logRequest(request)
    })
}
```

Does that make sense?

This approach has its own benefits (which I do like),

- **Cleaner Management**: Breaking out middleware into separate files makes it easier to manage and understand your codebase.
- **Prevents Conflicts**: By having a single main middleware file, you reduce the chance of conflicts between different middleware functions.

- **Optimized Performance**: Avoiding multiple layers of middleware can improve performance, as there are fewer steps for each request to go through.

Even though implementing advanced features such as socket implementation isn't supported directly in the **app** folder for API routes, Next.js offers everything necessary for developing APIs. This includes support for various HTTP methods

(**GET**, **POST**, **PUT**, **PATCH**, **DELETE**, **HEAD**, and **OPTIONS**), handling native Request and Response with **NextRequest** and **NextResponse**, headers, redirects, cookies, and even streaming.

New Next.js has got you all covered!

💥 **Surprise**

Now moving ahead, another thing that I would love to highlight is — **Cache**.

Yes, yet again!

Cache

By default, Next.js caches GET routes. So, if you keep seeing the same results even after adding new data through the API, don't worry, and don't blame Next.js. That's just how it operates.

If you would like to disable that feature, you can add,

```
export const dynamic = 'force-dynamic'

export async function GET() {}
```

Revalidation

You can also utilize the **`revalidate`** option for time-based validation to determine when you want to revalidate an API route.

```
export const revalidate = false

export async function GET() {}
```

Different Runtimes

Another interesting configuration available in Next.js is choosing the runtime environment. By default, Next.js operates on the Node.js runtime. However, we can switch to the edge runtime at the route level like this,

```
export const runtime = "edge"

export async function GET() {}
```

In the following chapter, you'll learn the difference between these two run times and when to use each of them.

For now, let's do the usual,

Tasks 🎯

📖 Develop a CRUD API application using Next.js's app directory.

📖 Explain the concept of middleware chaining in Next.js and provide strategies for effective management

📖 How would you implement nested dynamic routes with multiple dynamic segments in Next.js?

📖 What are the main differences between traditional backend development using Express.js and backend development with Next.js? Briefly outline the distinctions in architecture, routing, middleware usage, and API endpoint handling between the two frameworks

📖 Consider a scenario where you're tasked with developing a backend for a large-scale e-commerce platform. Given the choice between using Next.js and traditional Express.js for the backend development, which would you choose and why?

CHAPTER 12

Node vs Edge Runtime

In this chapter, we explore the differences between Node.js and Edge runtimes within the context of Next.js applications.

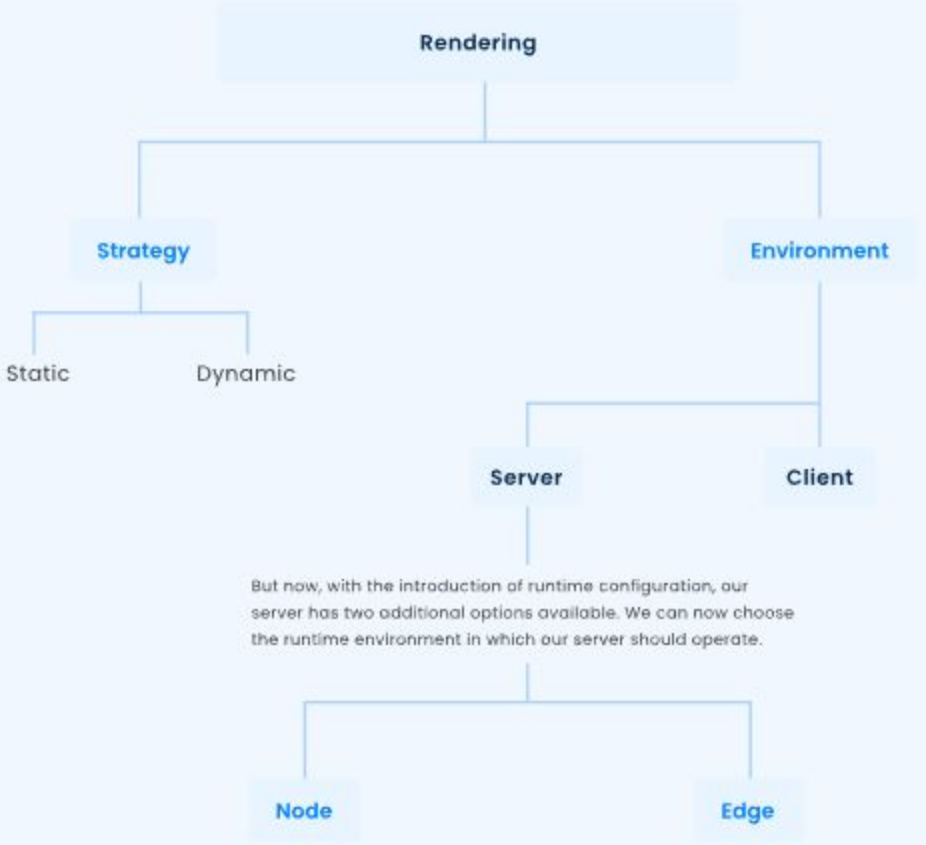
You'll gain insights into their distinct features, use cases, limitations, and how to leverage them effectively for optimizing application performance and user experience.

Node vs Edge Runtime

Now it's time to talk about something interesting, i.e., edge vs. node runtime.

By far, we learned different rendering strategies and environments in which we can render our application.

A quick map of that would be something like this



Edge Runtime

Edge runtime is like having a mini-computer close to where you are or where your devices are. Instead of doing everything on a faraway big computer, it does some tasks nearby.

This makes things faster, especially for stuff like websites, smart devices, or anything that needs quick responses. It's like having a helper right next to you instead of calling someone far away for every little thing.

So, how does edge compare with node.js runtime?

	Edge Runtime	Node Runtime
Location	Runs at the edge of the network, closer to users.	Typically runs on servers, cloud, or local machines
Purpose	Specialized for quick and efficient execution of code in distributed locations.	General-purpose runtime for executing JavaScript code.
Implementation	For example, Vercel's Edge Runtime, (is) built on the high-performance V8 JavaScript and WebAssembly engine.	Uses the Node.js environment, which is widely used for server-side scripting and development.
Use Cases	Geared towards low-latency scenarios, content delivery, and quick processing of tasks near users.	Versatile for a wide range of applications, from web servers to command-line tools, and server-side JavaScript applications.

While there are specific use cases for both, as usual, both have some limitations.

Limitations of Edge Runtime

- **Constrained Environment** - Edge Runtimes are often more constrained compared to full server environments, limiting certain capabilities.
- **Reduced Flexibility** - The specialized nature of Edge Runtime may limit the range of applications compared to more general-purpose runtimes.
- **Dependency on Network** - Since Edge Runtimes are distributed globally, they depend on network conditions, and issues may arise in case of network disruptions.

Limitations of Node Runtime

- **Potential Cold Starts**: Serverless implementations of Node.js may have cold start times, causing a slight delay when a function is invoked for the first time.
- **Resource Consumption**: While Node.js is lightweight, resource consumption can still be a consideration, especially in scenarios with limited resources.
- **Less Ideal for Ultra-Low Latency**: In situations where ultra-low latency is critical, the centralized nature of Node.js runtime may introduce some latency compared to Edge Runtimes.

Both have their advantages and disadvantages. And if I have to suggest something, then I would suggest, doing things depending on your needs.

Vercel, who has built its own edge runtime, shares a nice difference between all these and which features do edge and node support in Next.js applications:

Rendering: Edge and Node.js Runtimes

Link to docs ↗

Moving on, let's understand what edge and serverless functions are and when to use them.

Edge Function

An edge function is a piece of code that runs on servers at the edge of a network, closer to the user. It is a smart and quick responder. It does its job near you, making things faster, especially for applications that need immediate actions, like handling website requests or managing data from nearby devices.

Let's see a quick difference between edge vs serverless functions

	Edge Functions	Serverless Functions
Location	Run closer to the user, at the edge of the network	Typically run in a centralized cloud server.
Execution	Use specialized runtimes like Vercel's Edge Runtime	Use cloud providers' serverless computing services (e.g., AWS Lambda, Azure Functions, Vercel).

	Edge Functions	Serverless Functions
Startup	Faster startup times, eliminating cold boot delays	Might have some cold boot delays as the environment is set up. (If you don't use them for long, they'll go to their default inactive set)
Deployment	Deployed across a global network of data centers	Deployed on cloud servers managed by the service provider
Use Cases	Ideal for low-latency scenarios, delivering dynamic content quickly.	Versatile, suitable for various applications, especially those not requiring ultra-low latency.

In Next.js, we can decide the runtime of our choice in any layout, page, or even route. Switching between runtime is as easy as changing TV channels with a remote control. Too dreamy, isn't it?

But that's 100% true. You get the power of whatever you want to do and wherever you want to do with Next.js and Vercel powers. You just have to be a good developer writing better code.

So how can we change runtime in Next.js?

By default, Next.js uses node runtime. To switch to edge, all we have to do is specify this

```
export const runtime =  'edge' // 'nodejs' (default) | 'edge'
```

In any layout, page, or route. Isn't that super easy?

Demo

Now let's try it out inside code and see how it goes. We'll create two simple API routes with and without edge having the same content.

Node Route

app/api/node/route.ts

```ts
import { NextResponse } from 'next/server';

export async function GET() {
  // purposefully delay the response by 1 second
  await new Promise((resolve) => setTimeout(resolve, 1000));

  return NextResponse.json(
    { name: 'Hello from Node Runtime' },
    { status: 200 },
  );
}
```

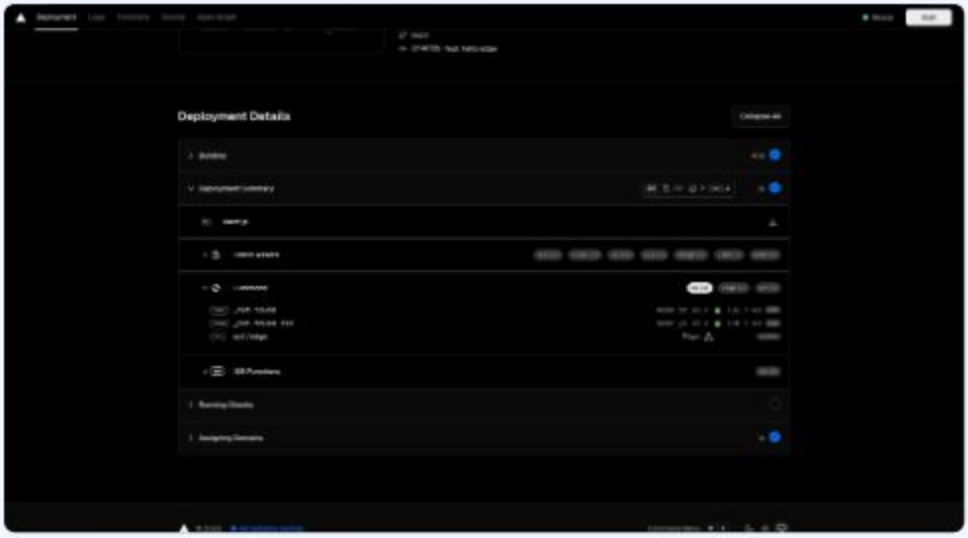
Edge Route

app/api/edge/route.ts

```ts
import { NextResponse } from 'next/server';

export const runtime = 'edge';

export async function GET() {
  // purposefully delay the response by 1 second
  await new Promise((resolve) => setTimeout(resolve, 1000));

  return NextResponse.json(
    { name: 'Hello from Edge Runtime' },
    { status: 200 },
  );
}
```

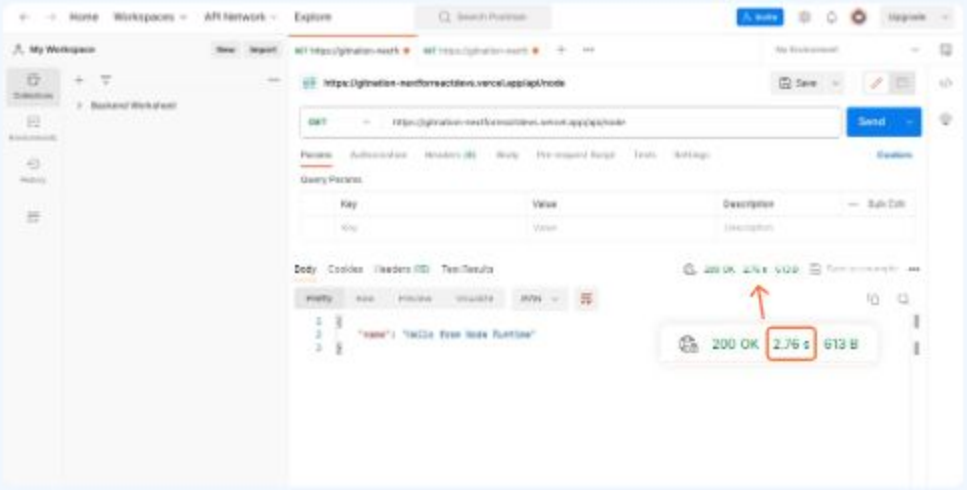
That's it. Let's deploy the application on Vercel to see what happens.

If we go to the deployments and open the deployment summary, we'll see **/api/edge** highlighted as an edge function there.

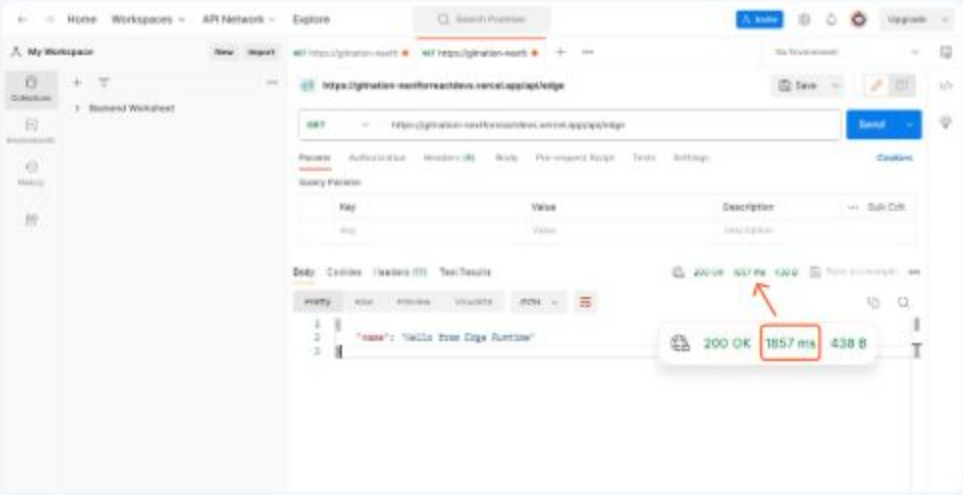


Perfect. Now let's test the speed or performance of both routes.

Make a request to **/api/node**, and we get the results and executing time for it is **2.76s**.



Doing the same for **/api/edge** gives us results in **1857ms**.



This is because Edge runtime is located on a server that's near to me. On the other hand, node runtime is located in a server used by Vercel at deployment. It could be far away from me as compared to edge runtime

That's good. So, **should we use edge over node runtime?**

Not at all. Edge has its limitations. And these include,

- **Quick, But Not Infinite**: Edge Functions need to respond within 25 seconds initially. So, they're like speedy helpers, but they can't take forever to do their job.
- **Watch the Size**: These functions can handle up to 4 MB of data, including all the code. It's like having a small backpack—fit what you need, but not everything.

- **Mind the Distance**: If your data source is far from where the Edge Function works, it might take a bit longer. It's like asking a friend to do something, but they need to travel a bit first. You can tweak settings to reduce this travel time.
- **Not All Tricks Allowed**: Some cool tricks that regular JavaScript knows, Edge Functions might not. It's like having a special toolbox; not all tools are there. You can check a list to see which APIs work for Edge here

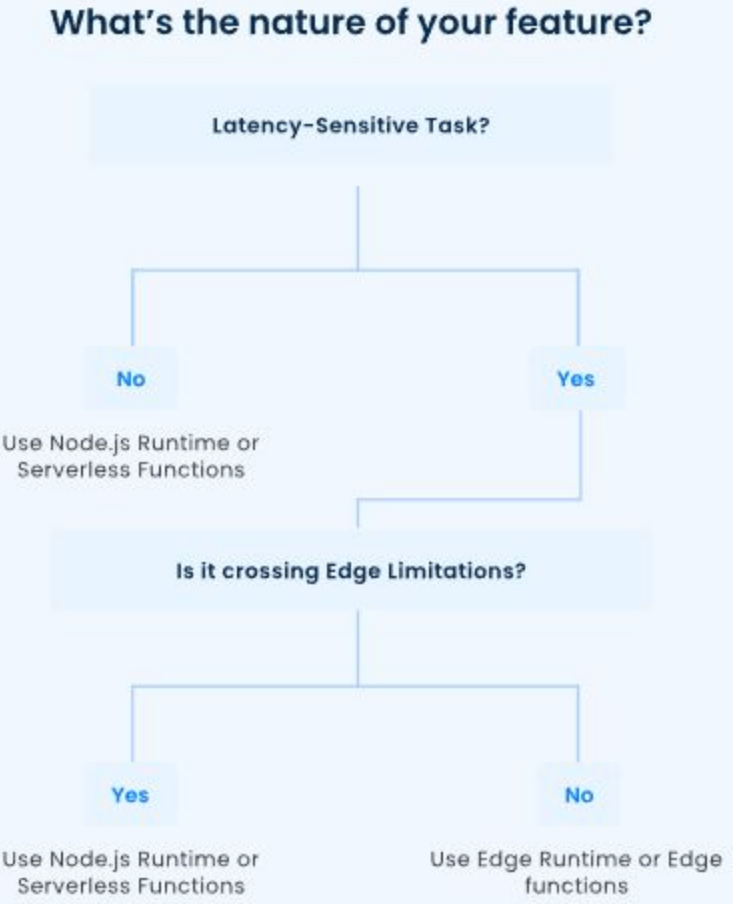
Rendering: Edge Runtime

Link to docs ↗

Okay, that's clear, but then when should we consider using edge over node runtime?

Here is a small mindmap that I use for myself. Ask yourself —

Here is a small mindmap that I use for myself. Ask yourself —



Always remember to check specific use case requirements and constraints to make an informed dev decision.

Use Case

Few use case examples of Edge Runtime

Notifications

Edge functions can be used to push real-time notifications to users based on their location or actions, providing instant updates without relying on a centralized server.

Authentication and Authorization

Handling user authentication and authorization at the edge can reduce latency for login processes, enhancing the user login experience.

Geolocation Services

Implement location-based services, such as geolocation tracking or local recommendations, by processing location data at the edge.

... and similar other specific use case. Edge functions are also often used in CDNs to deliver web content like images, videos, and scripts from servers closer to users. This reduces loading times for websites, improving the overall user experience. We learned that, don't we?

That's it about Edge vs. Node Runtime, Edge vs. Serverless functions, when to use them, how to use them, and everything else.

Tasks 🎯

- What are the fundamental differences between edge runtime and node.js runtime?
- Explain the limitations associated with edge runtime as outlined in the text. How might these limitations impact development decisions?
- In what scenarios might developers prefer node.js runtime over edge runtime, considering their respective advantages and limitations?
- How can we switch between edge and node.js runtimes in Next.js applications? What implications does this flexibility offer in terms of application development?
- You are a developer tasked with optimizing the performance of a real-time notification system for a social media platform.

  The current system experiences delays in delivering notifications to users, particularly those in regions with high network latency.

  Your goal is to improve the delivery speed of notifications to enhance user experience. How would you do that?

Onto the next now!

CHAPTER 13

Server Actions

In this chapter, we dive into the concept of Server Actions in Next.js, exploring their purpose, implementation, advantages, limitations, and practical use cases.

By the end, you'll have a comprehensive understanding of how Server Actions can streamline backend operations within your Next.js apps.

Server Actions

Maybe one of the hot topics of the recent tech world 🔥

Have you ever seen this before "`use server`" in any kind of Next.js application?

You might had a gut feeling that this is to symbolize whatever is written inside that file is treated as server things. And you're absolutely right!

Normally they're known as React Actions or React Server Actions. These are simple asynchronous functions that run on the server. That's it!

But why do we need server actions?

Although I wouldn't say it as a replacement but instead as an option to do mutations over the traditional route handlers.

We can use these server actions in both server and client components which makes them a handy feature to use. While Vercel might be inclined to showcase the use of this feature for form submissions or mutations, we can use them to query as well.

Aren't convinced yet? Let's see a demo, and we'll see how useful and handy they are while building an application.

We can create and use server actions in two ways,

Server Components

We add a **"use server"** directive of React inside any function, make it async, and turn it into a server action

Client Components

The way of using server actions inside client components is a little different.

Instead of what we did in the server component, we now have to create a new file, declare everything inside it as **"use server"** and import those functions inside the client component

Client Components

```
'use server';

export async function create() {
  // ...
}

export async function remove() {
  /// ...
}
```

And then call them wherever we want it to be. Be that be a client component or server component

```
import React, { useState } from 'react';
import { create } from './action'

const MyForm = () => {
  const [formData, setFormData] = useState({
    name: '',
    email: '',
  });

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value,
    });
  };

  const handleSubmit = (e) => {
    e.preventDefault();

  // call server action
  await create(formData)
  };
```

```
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input
          type="text"
          name="name"
          value={formData.name}
          onChange={handleChange}
        />
      </label>
      <br />
      <label>
        Email:
        <input
          type="email"
          name="email"
          value={formData.email}
          onChange={handleChange}
        />
      </label>
      <br />
      <button type="submit">Submit</button>
    </form>
  );
};

export default MyForm;
```

And as you might be wondering — Yes, we don't need to create any API to perform any normal CRUD. Server Actions have you covered. Next.js has made things that simple to do.

But now you might think — **How do they work? How are we able to call a server function right inside a client component?** 😎

Think! **How is it possible?**

No rocket science here. Server actions we use in client components are basically turned into POST requests by Next.js for us. For everything.

Any kind of server actions, be that for getting some information, adding some data to the database, updating or deleting it, Next.js calls our server through POST request only. But that is for server actions inside client components only

For server components, everything is on the server, i.e., component and server function, so they are executed right there, and we get HTML as a streamed response.

Now let's create a small application to see server actions in action.

Demo

We'll create a completely interactive, server-side form using the HTML form element that works even if javascript is disabled

```
function Home() {
  async function handleSubmit(event: FormData) {
    'use server';

    const data = {
      name: event.get('name'),
      email: event.get('email'),
    };

    console.log(data);
  }

  return (
    <main className="max-w-5xl p-24 flex flex-col mx-auto gap-10">
      <h1 className="font-bold text-2xl">Server Form</h1>
      <form action={handleSubmit} className="flex flex-col gap-5">
```

```
        <input
          type="text"
          name="name"
          className="p-2 text-zinc-800"
          placeholder="Your name"
        />
        <input
          type="email"
          name="email"
          className="p-2 text-zinc-800"
          placeholder="Your email"
        />
        <button type="submit" className="bg-teal-400 p-2 outline-
teal-800">
          Submit
        </button>
      </form>
    </main>
  );
}

export default Home;
```

For showing, if a form is submission process or not letting the user keep clicking on the form button while it's processing, we can show the form status and disable the button while a request is being made.

To do that, let's create another page inside this same application and see how we can show the form status

We'll use the same code as we did on the home page, but now we have to separate the button component from the main form as we'll have to turn it into a client component in order to use a specific form hook that React provides us.

It's called **useFormStatus** hook. It gives us status information on the last form submission.

useFormStatus – React

Link to docs ↗


```
import Button from "./button";

// a demo to show how to do loading and error handling with server
actions in a form
function Interactive() {
  async function handleSubmit(formData: FormData) {
    "use server";

    const data = {
      name: formData.get("name"),
      email: formData.get("email"),
    };

    // wait for a few milliseconds to simulate a slow server
    await new Promise((resolve) => setTimeout(resolve, 2000));

    console.log(data);
  }

  return (
    <main className="max-w-5xl p-24 flex flex-col mx-auto gap-10">
      <h1 className="font-bold text-2xl">Interactive Server Form</
h1>
      <form action={handleSubmit} className="flex flex-col gap-5">
```

```
        <input
          type="text"
          name="name"
          className="p-2 text-zinc-800"
          placeholder="Your name"
        />
        <input
          type="email"
          name="email"
          className="p-2 text-zinc-800"
          placeholder="Your email"
        />
        <Button />
      </form>
    </main>
  );
}

export default Interactive;
```

We can use that hook in our custom **Button** component like this

```
'use client';

import { useFormStatus } from 'react-dom';

function Button() {
  const { pending } = useFormStatus();

  return (
    <button
      type="submit"
      className="bg-teal-400 p-2 outline-teal-800"
      disabled={pending}
    >
      {pending ? 'Submitting...' : 'Submit'}
    </button>
  );
}

export default Button;
```

Similarly, we can also do error management using another special hook from React called **`useFormState`**. It allows us to update the state based on the result of a form action. You can learn more about it here

useFormState – React

Link to docs ↗

Or see the example in action here

Example

Link to example ↗

Now you might be wondering, can we not use server actions in a client form? The way we typically build forms using state and everything? Of course, we can

Let's try that too!

But now, remember, we're making the whole form as a client component, so, in order to use server action, we have to create a separate file and export it from there. And then, we'll import it into the client component form

```
"use client";

import { useState } from "react";
import { submitForm } from "./action";

function Page() {
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  async function handleSubmit(event:
React.FormEvent<HTMLFormElement>) {
    event.preventDefault();

    try {
      setLoading(true);
      setError(null);

      const formData = new FormData(event.currentTarget);
      const data = {
        name: formData.get("name"),
        email: formData.get("email"),
      };

      await submitForm(data);
    } catch (error: any) {
      setError(error);
    } finally {
      setLoading(false);
    }
  }

  return (
    <main className="max-w-5xl p-24 flex flex-col mx-auto gap-10">
      <h1 className="font-bold text-2xl">Normal Form</h1>
      <form onSubmit={handleSubmit} className="flex flex-col gap-5">
        <input
          type="text"
          name="name"
          className="p-2 text-zinc-800"
          placeholder="Your name"
        />
```

```
        <input
          type="email"
          name="email"
          className="p-2 text-zinc-800"
          placeholder="Your email"
        />
        <button
          type="submit"
          disabled={loading}
          className="bg-teal-400 p-2 outline-teal-800"
        >
          {loading ? "Submitting..." : "Submit"}
        </button>
      </form>

      {error && (
        <div className="bg-red-500 text-white p-2 rounded-md">
          {error.message}
        </div>
      )}
    </main>
  );
}

export default Page;
```

and form action like this

```
'use server';

export async function submitForm(data: {
  name: FormDataEntryValue | null;
  email: FormDataEntryValue | null;
}) {
  // wait for few milliseconds to simulate a slow server
  await new Promise((resolve) => setTimeout(resolve, 2000));

  console.log(data);
}
```

That's it above server actions. While we may have used small examples to showcase such a powerful feature, you're absolutely free to do any kind of complex scenarios using server actions.

To show you what I mean, let me show you the repository of my Next.js course. In this course, we cover all the querying and complex mutations. We dive into tasks like fetching recommendations or conducting a global search, and we handle all of this through server actions.

Next.js Course Repository
Link to repository ↗

There had been some concerns all over the internet when Next.js made a statement that server actions are stable from Next.js 14 and onwards. And the issue was security.

This meme from the presentation was all over Twitter.

People got a bit confused, thinking SQL is now directly inside JSX. Some even remembered their PHP days and started making fun of it.

As I mentioned earlier, it's not actual code inside the component. Even if it looks like it, Next.js will turn it into a POST request behind the scenes.

And when it comes to security, you can use anything inside a server action to check the input first before making the actual DB operations. Here's an example:

```
const schema = z.object({
  name: z.string().nonempty(),
  email: z.string().email(),
});

export async function create({ data }) {
  try {
    // check session
    const session = await user.getSession();

    if (!session) {
      throw new Error('Not authenticated');
    }

    const validatedFields = schema.safeParse(data);

    // Return early if the form data is invalid
    if (!validatedFields.success) {
      throw {
        errors: validatedFields.error.flatten().fieldErrors,
      };
    }

    // Your logic here
  } catch (error) {
    throw error; // Re-throw the caught error for further handling
  }
}
```

You can add as many security measures as you want to make it safe. It's in your hands!

Limitations

While server actions might seem cool to try—and they are—there are some drawbacks too:

- You can't use server actions for cross-platform development where you need the same backend or API on different ecosystems like mobile or desktop. Server action endpoints can't be exposed as of now.
- Server actions can't be used for scenarios like webhooks or sockets. In these cases, you need a server always listening for event updates, which server actions don't support since they're an integral part of the server and not exposed.
- Testing might feel challenging with server actions because you can't use tools like Postman. Testing has to be done directly. On the positive side, it enhances security since your API isn't exposed.
- For some, using POST requests for events like updates or deletes may not align with the best practices of HTTP protocols.

But overall, it's best to use if you want to build and ship applications faster. That's where they shine—in improving the developer experience.

CHAPTER 14

Styling

In this chapter, we'll explore various styling strategies available for Next.js applications. From traditional CSS approaches to utility-first frameworks like TailwindCSS, Next.js provides flexibility in choosing how you style your components.

Styling

So far we learned some cool core concepts of Next.js. Now coming back to the frontend again to see how we can make it better using different styling strategies.

We can absolutely use anything we want to style Next.js applications. Be that be plain CSS, SCSS, CSS module, tailwindcss, or even CSS in JS. It's compatible with everything.

Let's take a look at each of these one by one,

Plain CSS

It's straightforward. You write your CSS rules in separate files just like you normally would and refer those class/id names in components/ elements

It's straightforward. You write your CSS rules in separate files just like you normally would and refer those class/id names in components/ elements

styles.css

```css
.container {
  max-width: 960px;
  margin: 0 auto;
}

.title {
  font-size: 24px;
  color: #333;
}
```

page.js

```
import './styles.css';

const Component = () => {
  return (
    <div className="container">
      <h1 className="title">Hello, World!</h1>
    </div>
  );
};

export default Component;
```

Sass (Syntactically Awesome Style Sheets)

For using CSS with superpowers, i.e., Sass, we'll have to install the dedicated package for it,

It's straightforward. You write your CSS rules in separate files just like you normally would and refer those class/id names in components/ elements

```
npm install --save-dev sass
```

And then, we can go ahead and start styling with Sass

styles.scss

```scss
$primary-color: #007bff;

.container {
  max-width: 960px;
  margin: 0 auto;
}

.title {
  font-size: 24px;
  color: $primary-color;
}
```

page.js

```jsx
import './styles.scss';

const Component = () => {
  return (
    <div className="container">
      <h1 className="title">Hello, World!</h1>
    </div>
  );
};

export default Component;
```

Learn more about it here.

CSS Modules

CSS Modules provide local scoping for styles, which helps prevent style conflicts between components

page.module.css

```css
.container {
  max-width: 960px;
  margin: 0 auto;
}

.title {
  font-size: 24px;
  color: #333;
}
```

page.js

```jsx
import styles from './page.module.css';

const Component = () => {
  return (
    <div className="container">
      <h1 className="title">Hello, World!</h1>
    </div>
  );
};

export default Component;
```

While it's cool, you need to import styles for each component, which can be tedious and may clutter your codebase. Additionally, there's a learning curve for developers unfamiliar with this approach.

TailwindCSS

I can't recommend this enough. If you want my suggestion for styling, I'd pick Tailwind CSS every single time. Any day!

While I'd love to outline the steps for setting up Tailwind CSS, it's evolving. The anticipated v4 of Tailwind CSS is on its way, and it will bring changes to the setup process we currently use. You can find more information about it here.

However, just to mention it here, you can refer to this resource to learn how to set up Tailwind CSS in Next.js.

If you want to learn more about how Tailwind CSS works and why it's advantageous compared to other options, I highly recommend watching this crash course.

Tailwind CSS Full Course 2023 | Build and Deploy a Nike Website

Watch Now ↗

But you might think — **Which is the best option for styling?**

The choice between Plain CSS, Sass, CSS Modules, and TailwindCSS for styling depends on various factors such as project requirements, team preferences, scalability, and maintainability. Each option has its advantages and disadvantages:

Each option has its advantages and disadvantages:

Plain CSS

- Simple and easy to understand.
- Suitable for small projects or when you don't need complex styling.
- Might become hard to maintain in larger projects as the codebase grows.

Sass (Syntactically Awesome Style Sheets)

- Adds features like variables, mixins, and nesting to CSS.
- Helps in organizing and maintaining stylesheets.
- Offers better code reusability and maintainability compared to plain CSS.
- Requires a compilation step to convert Sass code into regular CSS.

CSS Modules

- Localizes CSS styles to specific components, reducing the chances of naming collisions.
- Encourages modular and reusable code.
- Automatically generates unique class names for each component, improving encapsulation.
- Works well with Next.js out of the box.

TailwindCSS

- Utility-first CSS framework that provides a set of pre-defined utility classes.
- Allows for rapid development by composing styles using utility classes.
- Promotes consistency across projects and reduces the need to write custom CSS.
- Might have a steeper learning curve for developers not familiar with utility-first approaches.

In the end, as I have always said, It's on you and your needs!

CHAPTER 15

More of Next.js

In this chapter, we'll explore additional advanced topics and resources related to Next.js, broadening our understanding and capabilities with this powerful framework.

More of Next.js

Up to this point, we've covered the fundamental concepts of Next.js, including its functionality, usage, and the new architecture it brings to both the frontend and backend. Now, let's dive into some additional topics.

Intercepting Routes

Intercepting routes is like redirecting a path in your application. Let's say you have different pages in your app, like a feed and a photo viewer. Normally, when you click on something in your feed to see a photo, the whole page might change to show the photo.

But intercepting routes means you can load just that part of the app without changing the whole page.

Meaning, instead of taking you to a separate page just for that photo, it pops up right there in the feed. That's intercepting routes in action. It's like the app catches that request to see the photo and shows it to you in a special way, without making you leave the feed.

Here is an **animated demo** that'll help you understand intercepting routes even better. Also, check **this example** by Vercel itself for you to try out.

You can check out the convention next.js suggests for creating Intercepting routes here.

Parallel Routes

Imagine you're building an app with different sections, like a dashboard with team info and analytics. Parallel routes let you show these sections at the same time or based on certain conditions, without changing the layout of the page.

Parallel routes work like that, showing multiple pages simultaneously or conditionally within your app.

Here is an **animated demo** showcasing how parallel routes work and how you can do the same

Database

Again, it depends on your needs. What you want to do and how well you want to d-. Several factors come into play, including the specific requirements of your project, your team's familiarity with different databases, performance considerations, scalability needs, and budget constraints.

Here is a nice article covering popular databases to use with Next.js, with links on how to use them in Next.js. Do try all!

Resources

Recently, I was invited to conduct a session on Next.js for GitNation, where I dive into the complexities of Next.js and its functionality within the context of RSC. I provided deeper insights into how Next.js operates. For a more comprehensive understanding, feel free to check it out.

Parallel routes work like that, showing multiple pages simultaneously or conditionally within your app.

Next.js for React.js Developers - GitNation

Watch Now ↗

... that may/may not be the end of the list. But for now, it is! Finally 😵

We talked about a lot of stuff regarding Next.js and how it can improve our application in different aspects. I hope this helps you become a better Next.js Developer

Happy coding!