
NEXT.JS

Hi! I am Mosh Hamedani. I'm a software engineer with over 20 years of experience and I've taught millions of people how to code and become professional software engineers through my YouTube channel and coding school (Code with Mosh).

This PDF is part of my **Next.js course** where you will learn everything you need to build full-stack applications with Next.js 13 and TypeScript.

https://codewithmosh.com

https://www.youtube.com/c/programmingwithmosh

https://twitter.com/moshhamedani

https://www.facebook.com/programmingwithmosh/

Table of Content

Getting Started

Terms

Client components
Client-side Rendering (CSR)
Dynamic rendering
Node.js runtime
Server components
Server-side Rendering (SSR)
Static rendering
Static Site Generation (SSG)

Summary

- Next.js is a framework for building fast, and search-engine friendly applications.
- It includes a compiler for transforming and minifying JavaScript code, a Command-line Interface (CLI) for building and starting our application, and a Node.js runtime for running backend code. This enables full-stack development.
- With Next.js, we can render our components on the server and return their content to the client. This technique is called *Server-side Rendering* (SSR) and makes our applications search-engine friendly.
- To further improve performance, we can pre-render pages and components that have static data during the build and serve them whenever needed. This technique is called *Static Site Generation* (SSG).
- The new app router in Next.js 13 makes it incredibly easy to create routes. We can define route segments by creating directories. To make a route public, we add a page file (page.js, page.jsx, or page.tsx) in the corresponding directory.

- Next.js provides the Link component to enable *client-side navigation*. This means as the user navigates between pages, the new content is loaded quickly and smoothly without the entire page being reloaded.

- Next.js 13 supports *client and server components* introduced in React 18. Client components are rendered on the client within a web browser. This technique is called *Client-side Rendering (CSR)* and it's how traditional React apps work. Server components are rendered on the server within a Node.js runtime. This technique is called *Server-side Rendering (SSR)*.

- Server components lead to reduced bundle sizes, better performance, increased search engine visibility, and enhanced security. But they cannot handle browser events, access browser APIs, or use the state or effect hooks. These functionalities are only available in client components. So we should use them whenever possible unless we need interactivity.

- All the components and pages in the /app directory are server components by default. To make a component a client component, we add the **'use client'** directive on top of the component file.

- Server components are great for fetching data because they don't require extra server trips, making our application faster and more search-engine friendly.

- Next.js enhances the fetch() function by adding automatic caching. This boosts performance and reduces the need to retrieve the same piece of data twice.

- In Next.js, components can be rendered at build time (called *Static Rendering*) or at request time (called *Dynamic Rendering*). If we have pages or components with static data, we can pre-render them during build time to improve our application's performance.

Key Commands

```
# Creating a new project
npx create-next-app

# Running the dev server
npm run dev

# Building the application
npm run build

# Starting the application in production mode
npm start
```


Styling

Terms

CSS modules
Daisy UI
Global styles
PostCSS
Tailwind

Summary

- In Next.js projects, we define global styles in **/app/global.css**. Reserve this file for global styles that need to be applied across multiple pages and components. Avoid adding excessive styles to this file, as it can quickly grow out of hand and become difficult to maintain.

- In traditional CSS, if we define the same class in two different files, one will overwrite the other depending on the order in which these files are imported. *CSS modules* help us prevent this problem. A CSS module is a CSS file that is scoped to a page or component.

- During the build process, Next.js uses a tool called *PostCSS* to transform our CSS class names and generate unique class names. This prevents clashes between different CSS classes across the application.

- *Tailwind* is a widely-used CSS framework for styling application. It offers a comprehensive set of small, reusable utility classes. We can combine these classes to create beautiful user interfaces.

- DaisyUI is a component library built on top of Tailwind. It provides a collection of pre-designed and reusable components such as accordion, badge, card, etc.

Routing and Navigation

Terms

Client cache
Dynamic routes
Layout
Prefetching

Summary

- The new App router in Next.js uses convention over configuration to define routes. It looks for special files such as page.tsx, layout.tsx, loading.tsx, route.tsx, etc.

- With the App router, we can colocate our pages and their building blocks (eg components, services, etc). This helps us better organize our projects as we can keep highly related files next to each other. No need to dump all the components in a centralized components directory.

- A *dynamic route* is one that takes one or more parameters. To add parameters to our routes, we wrap directory names with square brackets (eg [id]).

- In standard React applications, we use the state hook for managing component state. In server-rendered applications, however, we use query string parameters to keep state. This also allows us to bookmark our pages in specific state. For example, we can bookmark a filtered and sorted list of products.

- We use *layout* files (layout.tsx) to create UI that is shared between multiple pages. The root layout (/app/layout.tsx) defines the common UI for all our pages. We can create additional layouts for specific areas of our application (eg /app/admin/layout.tsx).

- To provide smooth navigation between pages, the Link component prefetches the links that are in the viewport.

- As the user moves around our application, Next.js stores the page content in a cache on the client. So, if they revisit a page that already exists in the cache, Next.js simply grabs it from the cache instead of making a new request to the server. The client cache exists in the browser's memory and lasts for an entire session. It gets reset when we do a full refresh.

File Conventions

```
page.tsx
layout.tsx
loading.tsx
not-found.tsx
erorr.tsx
```

Accessing Route and Query String Parameters

```tsx
interface Props {
  params: { id: string },
  searchParams: { sortOrder: string }
}

export default function Home({ params, searchParams }: Props) {
}
```

Programmatic Navigation

```tsx
const router = useRouter();
router.push('/')
```

Building APIs

Terms

API endpoint
Data validation libraries
HTTP methods
HTTP status codes
Postman
Route handlers

Summary

- To build APIs, we add a route file (route.tsx) in a directory. Note that within a single directory, we can either have a page or a route file but not both.
- In route files, we add one or more route handlers. A *route handler* is a function that handles an HTTP request.
- HTTP requests have a *method* which can be GET (for getting data), POST (for creating data), PUT/PATCH (for updating data), and DELETE (for deleting data).
- HTTP protocol defines standard status codes for different situations. A few commonly used ones include: 200 (for success), 201 (when a resource is created), 400 (indicating a bad request), 404 (if something is not found), and 500 (for internal server errors).
- To create an object, the client should send a POST request to an API endpoint and include the object in the body of the request.
- We should always validate objects sent by clients. We can validate objects using simple if statements but as our applications get more complex, we may end up with complex and nested if statements.

- Data validation libraries, such as Zod, allow us to define the structure of our objects using a simple syntax, taking care of all the complexity involved in data validation.

- To update an object, the client should send a PUT or PATCH request to an API endpoint and include the object in the request body. PUT and PATCH are often used interchangeably. However, PUT is intended for replacing objects, while PATCH is intended for updating one or more properties.

- To delete an object, the client should send a DELETE request to an API endpoint. The request body should be empty.

- We can use Postman for testing APIs. With Postman we can easily send HTTP requests to API endpoints and inspect the responses.

Creating Route Handlers

```
export async function GET(request: NextRequest) {
  const body = await request.json();

  return NextResponse.json(body, { status: 200 });
}
```


Database Integration

Terms

Databases
Database Engines
Migrations
Models
Object-relational Mapper (ORM)
Prisma

Summary

- We use *databases* to permanently store data. There are many *database engines* available. Some of the popular ones are MySQL, PostgreSQL, MongoDB, etc.

- To connect our applications to a database, we often use an *Object-relational Mapper* (ORM). An ORM is a tool that sits between a database and an application. It's responsible for mapping database records to objects in an application. Prisma is the most widely-used ORM for Next.js (or Node.js) applications.

- To use Prisma, first we have to define our data *models*. These are entities that represent our application domain, such as User, Order, Customer, etc. Each model has one or more fields (or properties).

- Once we create a model, we use Prisma CLI to create a *migration* file. A migration file contains instructions to generate or update database tables to match our models. These instructions are in SQL language, which is the language database engines understand.

- To connect with a database, we create an instance of PrismaClient. This client object gets automatically generated whenever we create a new migration. It exposes properties that represent our models (eg user).

Key Commands

```
# Setting up Prisma
npx prisma init

# Formatting Prisma schema file
npx prisma format

# Creating and running a migration
npx prisma migrate dev
```

Working with Prisma Client

```
await prisma.user.findMany();
await prisma.user.findUnique({ where: { email: 'a' } });
await prisma.user.create({ data: { name: 'a', email: 'a' } });
await prisma.user.update({ where: { email: 'a' }, data: { email: 'b' } });
```


Uploading Files

```
# Setting up cloudinary
npm i next-cloudinary

# Setting the env variable
NEXT_PUBLIC_CLOUDINARY_CLOUD_NAME="..."
```

Uploading Files

```
<CldUploadWidget
  uploadPreset="..."
  options={{
    sources: ['local'],
    maxFiles: 5,
    multiple: false,
    styles: {},
  }}
>
  {({ open }) => (
    <button onClick={() => { open() }}>Upload</button>
  )}
</CldUploadWidget>
```

Authentication

Terms

Authentication session
Database adapter
JSON Web Token (JWT)
Middleware
Next Auth

Summary

- NextAuth.js is a popular authentication library for Next.js applications. It simples the implementation of secure user authentication and authorization. It supports various authentication providers (eg Google, Twitter, GitHub, Credentials, etc).

- When a user signs in, Next Auth creates an *authentication session* for that user. By default, authentication sessions are represented using *JSON Web Tokens* (JWTs). But sessions can also be stored in a database.

- To access the authentication session on the client, we have to wrap our application with SessionProvider. This component uses React Context to pass the authentication session down the component tree. Since React Context is only available in client components, we have to wrap SessionProvider with a client component.

- Using *middleware* we can execute code before a request is completed. That's an opportunity for us to redirect the user to the login page if they try to access a private part of our application without having a session. Next Auth includes built-in middleware for this purpose.

- Next Auth comes with many *database adapters* for storing user and session data.

Setting Up Next Auth

```
export const authOptions: NextAuthOptions = {
  providers: [
    GoogleProvider({
      clientId: process.env.GOOGLE_CLIENT_ID!,
      clientSecret: process.env.GOOGLE_CLIENT_SECRET!,
    })
  ],
  adapter: PrismaAdapter(prisma),
  session: {
    strategy: 'jwt'
  }
};

const handler = NextAuth(authOptions);

export { handler as GET, handler as POST };
```

Auto-generated Endpoints

```
# Sign in
/api/auth/signin

# Sign out
/api/auth/signout
```

Protecting Routes

```
// middleware.ts
export { default } from 'next-auth/middleware';
export const config = { matcher: ["/admin/:path*"] }
```

Accessing the Session

```
// On the client
const { data: session, status } = useSession();

// On the server
const session = await getServerSession(authOptions);
```


Sending Emails

Setting Up React Email

```
npm i react-email @react-email/components
```

Creating a Template

```
import { Html, Body, Container, Text } from '@react-email/components';

const WelcomeTemplate = () => {
  return (
    <Html>
      <Body>
        <Container>
          <Text>Hello World</Text>
        </Container>
      </Body>
    </Html>
  );
};
```

Sending an Email

```
import { Resend } from 'resend';

const resend = new Resend('api-key');

await resend.emails.send({
  from: '...',
  to: '...',
  subject: '...',
  react: <WelcomeTemplate />
})
```

Optimizations

Terms

Image component
Metadata
Lazy loading
Link component
Script component

Summary

- The Image component in Next.js automatically optimizes and serves images in various formats and sizes, reducing loading times and bandwidth usage for improved site performance.
- The Link component enables client-side navigation between pages within our Next.js application, eliminating full page reloads and creating a smoother user experience.
- The Script component allows you to load and manage external scripts efficiently.
- Next.js automatically optimizes our fonts and removes external network requests for improved privacy and performance.
- To make our applications search engine friendly, we can export a metadata object from our pages and layouts. Metadata exported from a page overwrite metadata defined in the layouts.
- Lazy loading helps us improve the initial loading performance of a page by reducing the amount of JavaScript needed to render the page. With lazy loading we can defer loading of client components and external libraries until when they're needed.

Creating Metadata

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


export async function generateMetadata(): Promise<Metadata> {

};
```

Lazy Loading

```tsx
import dynamic from "next/dynamic";
const HeavyComponent = dynamic(
  () => import('./HeavyComponent'),
  {
    ssr: false,
    loading: () => <p>Loading...</p>
  }
);
```

