Next.js Cookbook

Learn how to build scalable and high-performance apps from scratch

Andrei Tazetdinov

bpb

Next.js Cookbook

Learn how to build scalable and high-performance apps from scratch

Andrei Tazetdinov

www.bpbonline.com

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-453

www.bpbonline.com

Dedicated to

My beloved wife:
Eugenie
&
My Daughter Alice

About the Author

Andrei Tazetdinov, is a highly experienced software engineer with over 16 years of experience in the industry. Currently working at IBM IX as a Senior Frontend Developer and Technical Architect, Andrei Tazetdinov has a passion for creating innovative and user-friendly applications using cutting-edge technology.

Throughout Andrei Tazetdinov's career, he has worked on numerous projects across various industries, ranging from healthcare to finance. Their experience has given them a deep understanding of the software development process, from ideation to deployment.

In addition to his professional work, Andrei Tazetdinov is also passionate about sharing their knowledge with others. He was a teacher at Samsung Coding School for several years and worked with teenagers to guide them and create their very first Android applications using Java.

With this book, Andrei Tazetdinov hopes to help aspiring developers and experienced professionals alike to become proficient in building full-stack applications using NextJS. His wealth of knowledge and experience in the field makes him the perfect guide for anyone looking to start or advance their career as a full-stack developer.

Thank you for choosing this book

About the Reviewer

Denis Bezrukov is a dedicated Frontend Developer with a rich professional background in the field of Forex trading platforms. His experience spans 4 years, during which he has cultivated a robust knowledge baseD and skill set in the development of complex applications utilizing React, Redux.

As a core contributor to the Rome tool project, a linter and formatter written in Rust, Denis has honed his ability to create modern web developers tools.

Acknowledgements

I would like to express my heartfelt gratitude to my family and friend, who have always been my pillars of strength and support. Their unwavering love and encouragement have helped me navigate through the ups and downs of life, and have been instrumental in shaping me into the person I am today.

In particular, I am immensely grateful to my daughter Alice, whose boundless curiosity and infectious energy have been a constant source of inspiration for me. Her insatiable thirst for knowledge and her relentless pursuit of excellence have reminded me of the importance of curiosity and determination, and have challenged me to strive for greatness in my own work.

I am also indebted to my colleagues, mentors, and friends, who have generously shared their time, expertise, and insights with me throughout my journey. Their wisdom, guidance, and constructive feedback have helped me refine my ideas, sharpen my skills, and broaden my horizons. I want to gratefully thank Denis Bezrukov from JetBrains for his support and help with this book review.

Finally, I would like to express my appreciation to all the readers of this book, whose interest and enthusiasm have motivated me to share my knowledge and insights with the world. It is my hope that this book will inspire and inform and that it will contribute to a deeper understanding and appreciation of the topics it explores.

Preface

As a full-stack developer, you need to master a variety of programming languages, frameworks, and tools to build robust, scalable, and user-friendly web applications. In recent years, NextJS has emerged as one of the most popular and powerful frameworks for building server-side-rendered React applications. With its intuitive API, powerful features, and vibrant community, NextJS has become the go-to choice for developers who want to create high-performance web applications with ease.

This book is designed to help you get started with NextJS and take your full-stack development skills to the next level. Whether you are a seasoned developer looking to expand your skillset or a newcomer to the world of web development, this book will provide you with the knowledge, tools, and techniques you need to build modern, dynamic web applications that meet the needs of today's users.

In this book, we will cover a wide range of topics, including:

- The basics of NextJS and its core features
- How to create and configure a NextJS application from scratch
- The power of server-side rendering and its benefits
- Best practices for styling, routing, and data fetching with NextJS
- Advanced topics such as testing, optimization, and deployment
- AWS Amplify as a hosting provider and many more topics

We will also provide you with plenty of hands-on examples, practical exercises that will help you improve your skills and confidence as a full-stack developer. By the end of this book, you will be able to create sophisticated web applications that leverage the power of NextJS and React, and you will be well on your way to a successful career in full-stack development.

So, let's get started and explore the exciting world of NextJS!

Chapter 1: Warming up with NextJS - In this chapter, readers will be introduced to NextJS and will learn how to set up their development environment. They will be guided through the installation of the necessary software and tools, and will

learn how to create a new NextJS application from scratch. Readers will also learn the basics of NextJS, including how to work with the NextJS file system, how to create pages and components, and how to use the built-in routing system. By the end of this chapter, readers will have created their first NextJS application and will be ready to dive deeper into the framework's features and capabilities.

Chapter 2: Using design patterns in NextJS - In this chapter, readers will learn how to use design patterns to optimize their NextJS application development. The chapter will cover several common design patterns, including the Singleton pattern, the Strategy pattern, and the Builder pattern. Readers will learn how these patterns can be applied to NextJS to improve the efficiency and scalability of their applications. The chapter will also provide practical examples of how to implement these patterns in NextJS, with step-by-step instructions and code snippets. By the end of this chapter, readers will have a solid understanding of how to apply design patterns in NextJS and will be able to create more robust and efficient web applications.

Chapter 3: Authorization in a glance with NextJS - In this chapter, readers will learn about authorization in NextJS and how to implement it in their applications. The chapter will cover the basics of authentication and authorization, and will provide an overview of different authentication methods that can be used in NextJS. Readers will also learn how to create a basic authorization system using NextJS's built-in authentication features. Additionally, the chapter will cover best practices for securing user data and preventing unauthorized access to sensitive information. By the end of this chapter, readers will have a solid understanding of how to implement authorization in NextJS and will be able to create secure and reliable web applications.

Chapter 4: Server-side power of NextJS - In this chapter, readers will learn about the server-side rendering features of NextJS and how to take advantage of them in their applications. The chapter will cover the benefits of server-side rendering, including improved performance, SEO, and user experience. Readers will also learn how to set up and configure server-side rendering in NextJS, as well as how to work with dynamic data and API calls in a server-side rendered application. Additionally, the chapter will cover best practices for optimizing server-side

rendered applications and handling errors. By the end of this chapter, readers will have a solid understanding of how to use server-side rendering in NextJS to create fast, dynamic, and highly scalable web applications.

Chapter 5: Using state management in NextJS - In this chapter, readers will learn about state management in NextJS and how to implement it using the popular Redux library. The chapter will cover the basics of state management, including why it's important and how it works. Readers will also learn how to set up and configure Redux in NextJS, as well as how to work with Redux actions, reducers, and stores. Additionally, the chapter will cover best practices for optimizing state management in NextJS, including how to handle asynchronous actions and how to use middleware. By the end of this chapter, readers will have a solid understanding of how to use state management with Redux in NextJS and will be able to create complex and dynamic web applications with ease.

Chapter 6: Implementing internal pages using NextJS - In this chapter, readers will learn how to create internal pages in NextJS and how to implement a basic CRUD system for managing user data. The chapter will cover the basics of creating dynamic pages in NextJS, including how to work with dynamic routes and how to pass data between pages. Readers will also learn how to set up and configure a database, and how to use NextJS's built-in API routes to handle requests and responses. Additionally, the chapter will cover best practices for optimizing internal pages in NextJS, including how to use caching and how to handle errors. By the end of this chapter, readers will have a solid understanding of how to create and manage internal pages in NextJS and will be able to build complex and powerful web applications.

Chapter 7: The superpower of E2E testing in NextJS - In this chapter, readers will learn about end-to-end (E2E) testing in NextJS and how to implement it using the Cypress and Playwright testing frameworks. The chapter will cover the basics of E2E testing, including why it's important and how it works. Readers will also learn how to set up and configure Cypress and Playwright in NextJS, as well as how to write and run tests for a NextJS application. Additionally, the chapter will cover best practices for optimizing E2E testing in NextJS, including how to handle asynchronous requests and how to use test-driven development principles. By the

end of this chapter, readers will have a solid understanding of how to use E2E testing with Cypress and Playwright in NextJS and will be able to create robust and reliable web applications.

Chapter 8: Deploying NextJS project to production - In this chapter, readers will learn how to deploy their NextJS application to production using the AWS Amplify platform. The chapter will cover the basics of deployment, including why it's important and how it works. Readers will also learn how to set up and configure their AWS Amplify account, and how to connect their NextJS application to the platform. Additionally, the chapter will cover best practices for optimizing deployment in NextJS, including how to use environment variables. By the end of this chapter, readers will have a solid understanding of how to deploy their NextJS application to production using AWS Amplify and will be able to launch their application with confidence.

Chapter 9: Mastering optimization tools for NextJS - In this chapter, readers will learn about optimization tools for NextJS and how to use them to improve the search engine optimization (SEO) of their application. The chapter will cover the basics of SEO, including why it's important and how it works. Readers will also learn how to set up and configure optimization tools and NextJS Image Optimization. Additionally, the chapter will cover best practices for optimizing SEO in NextJS, including how to use metadata and structured data, and how to improve page speed. By the end of this chapter, readers will have a solid understanding of how to use optimization tools for NextJS to improve the SEO of their application and will be able to maximize their online visibility.

Coloured Images

Please follow the link to download the
Coloured Images of the book:

https://rebrand.ly/g7kjstb

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

Chapter 1
Warming up with NextJS

'The beginning is always today.'

— Mary Shelley

Introduction

Greetings, a future chef in the NextJS kitchen. In this chapter, we will begin our journey into the world of sophisticated software development using the most advanced web application framework to date.

I will not delve into the intricacies of the initial setup of the environment. If you are here and ready to cook real masterpieces, then the environment necessary for installing and configuring the **nodejs** is already on your PC (or Mac, depending on preferences of course).

Structure

- Setup and run NextJS
  - Install using npm and the latest version of NodeJS
  - For the older npm versions
    - How to run the project for local development?

- How to customize Webpack
  - How to use Typescript in NextJS?
  - How to use SCSS in NextJS
- How to create a multipage app?
- How to change pages. Routing tools
  - How to change page params state without running data fetching methods
- Conclusion

Objectives

In this chapter, we will install and set up the local development environment for the easy start of the NextJS application. After you finish this chapter, you will set up and run your new NextJS application and make as many configurations as you possibly need for the first start. You will learn how to create a multi-page application with the framework. Also, you will learn how to navigate between pages and how you can manage the router properties.

Setup and run NextJS

For better performance and stability, I recommend a version of the NPM module equal to 5.2+ or higher. You can also use the older **npm-install-way** if you prefer it for some reason or you cannot install the latest version of NodeJS

Let us check several ways to install and set up your first project with NextJS:

In this book we will use yarn as an alternative to npm. Yarn provides a number of features that are not available in npm, such as faster and more reliable dependency installation, improved network performance, and better security features. You can collect it using this link https://yarnpkg.com/

Installing using npm and the latest version of NodeJS

Just type the commands into your console. Please note that in this book we will use Typescript to produce the application. For the correct setup please use the commands as follow:

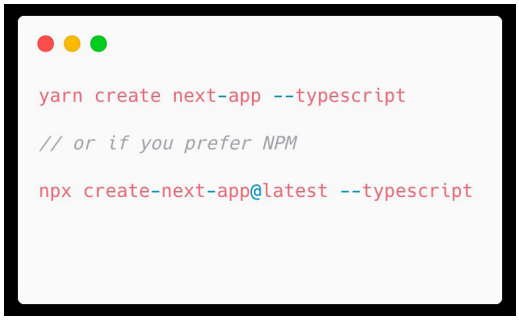


***Figure 1.1**: Commands to install NextJS*

After the installation is complete, your project will contain all files and configurations to quick-start your new project and learn NextJS.

create-next-app contains some other useful commands for project creation that will help you to understand how to use a framework. There is a possibility to use the GitHub URL as an example for your first application. The command should look like this:

yarn create next-app -example **https://github.com/vercel/next.js/tree/canary/examples/auth0**

In this case, you will create a blank project with Auth0 possibility inside with configured API and pages to a successful login. Please check this link for more examples: **https://github.com/vercel/next.js/tree/canary/examples**

If you have already set up your project this way you can skip the next section.

For the older npm versions

For the older npm versions, use these commands to start the new project.

After successful setup and creation of the project do not forget to add the Typescript to your project in case when you have created the project by this way:

***Figure 1.2**: Manual project creation*

To add the Typescript please follow these instructions

Create the file in the root of the project using your IDE or with the CLI command:

1. `touch tsconfig.json` for Linux/macOS
2. `echo > tsconfig.json` for Windows

NextJS will automatically do all required setup for the Typescript you will need only start your project using commands:

```
yarn dev or npm run dev
```

After that you will probably see answers about requirements for the project if you do not have them installed into your project yet eg. **@types/react, @types/node, @types/react-dom**. Just follow the instructions to complete the setup for your project

In the end, the whole setup will be complete. Please note several things:

You will see a file with this name in your root directory next-env.d.ts**. Do not remove it or change the file body for any reason.** It is auto-generated by the Typescript compiler. Please do not make any changes in the file next-env.d.ts use the instruction below instead. Your configuration file tsconfig.json contains information about the types that you will use in your project. Just add a new type file into the include section to use it

Find below the example of the `tsconfig.json` file that you should have as a result:

```
{
  "compilerOptions": {
    "target": "es5",
    "lib": [
      "dom",
      "dom.iterable",
      "esnext"
    ],
    "allowJs": true,
    "skipLibCheck": true,
    "strict": true, // do not forget to make it TRUE for the production
    "forceConsistentCasingInFileNames": true,
    "noEmit": true,
    "esModuleInterop": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "jsx": "preserve",
    "incremental": true
  },
  "include": ["next-env.d.ts","any-new-types.d.ts","**/*.ts","**/*.tsx"],
  "exclude": ["node_modules"]
}
```

***Figure 1.3**: Code in tsconfig.json file*

Why do we ever need this feature? Let us use imagination a bit to understand this feature.

The general difference between using the ***.ts** files and ***.d.ts** files is that the second one is used to declare a type definition. Still, not much clearance because we can use both file types to declare a type. Ok, how about if we could declare the existing ***JavaScript*** function for TypeScript? For example, we have a function like this in the **printHello.js** file:

***Figure 1.4**: Code in printHello.ts file*

And as a declaration in file **printHello.d.ts** will be the following:

***Figure 1.5:** Code in printHello.d.ts file*

Now we can use function **`printHello`** without any compilation errors as it is declared in the file with declarations.

If you made a setup in a modern and automatic way you can skip the next section and proceed with the first commands to start.

Next JS in general works with a page-oriented architecture so the basic element in the framework is a **page**. The page has its URL link from its creation. To create your first page please check that you have a **pages** folder in your root folder of the project. The main page (or the default page) will always be called the **`index.tsx`**. In the next sections we will figure out how to rewrite and redirect pages but the name of the default page is always **index.**

The next step is to create the file **index.tsx** in the pages folder with this code inside:

```
function MainPage() {
  return <div>Your cookbook with tasty recipes. Yammy</div>
}

export default MainPage
```

***Figure 1.6**: Code in index.tsx file*

Please check if you have these commands in your **package.json** file. If you do not have them then please create:

```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start"
}
```

***Figure 1.7**: Code in package.json file*

How to run the project for local development

Just run the command in your project root: **yarn dev** or **npm run dev**

After that, you can proceed with the URL from your CLI. By default, it is **http://localhost:3000**

How to customize WebPack

Let us initialize the problem of why we would ever need to customize the WebPack for our project.

Imagine that we have different behavior in development and production mode. For example, we can use different environments and secret keys for each kind of build. To see it we need it to inject some logic into the build process or create logic in components.

We do not recommend the implementation of such logic inside the components themselves and highly recommend separating this behavior in the build process

To make changes in WebPack for NextJS let us check what changes we can do in the NextJS configuration. That can be done using the **`next.config.js`** file.

As in the case with declaration configuration, we highly recommend using separated files for the environment variables that will be changed with the development process. So please create files like **`.env.local`** or **`.env.development`** to change variables while your project is in support conditions

For example, we can create the process variable that will be used in the development or production flow. The body of your env file should look like this:

***Figure 1.8**: Variable declaration in env file*

Your WebPack updates can be done in file **`next.config.js`**:

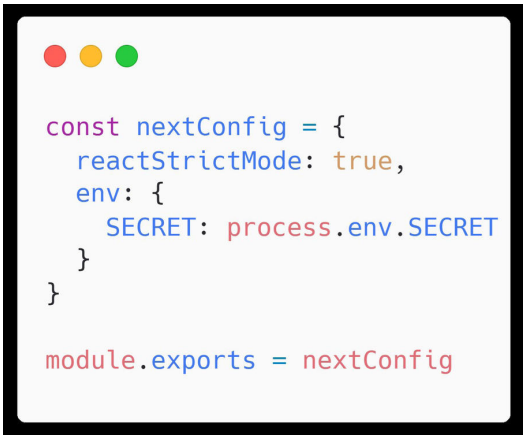


***Figure 1.9**: Code in next.config.js file*

As you can see, we have added the process variable that can be used in our project. Please note that adding or changing any variable should be followed with a development server restart (or production rebuild, depending on what flow do you currently use).

After that, you will be able to use your variable in the code like this or with any flow you wish

```
function SecretPage() {
  return <div>Your secret is {process.env.SECRET}. Do not tell it to anyone</div>
}

export default ExamplePage
```

***Figure 1.10**: Code in secret page file*

Now we can insert some logic into WebPack in our configuration to act only in development start, as an example:

```
const nextConfig = {
  reactStrictMode: true,
  styledComponents: true,
  webpack: (config, { buildId, dev, isServer, defaultLoaders, webpack }) => {
    if(dev) {
      console.log('Is development flow ', process.env.SECRET) // this line will fire twice because webpack function
is runs for server and client separately
    }
    return config // do not forget to return this object
  },
  env: {
    SECRET: process.env.SECRET
  }
}

module.exports = nextConfig
```

***Figure 1.11**: Code in next.config.js file*

You can inject any logic here or update the config with new plugins that you want to use like this:

```
const nextConfig = {
  reactStrictMode: true,
  styledComponents: true,
  webpack: (config, { buildId, dev, isServer, defaultLoaders, webpack }) => {
    if(dev) {
      console.log('Is development flow ', process.env.SECRET) // this line will fire twice because webpack function
is runs for server and client separately
    }

    const newConfig = config.plugins.push(<YOUR-PLUGINS-CONFIGURATIONS>)

    return newConfig // do not forget to return this object
  },
  env: {
    SECRET: process.env.SECRET
  }
}

module.exports = nextConfig
```

***Figure 1.12**: Code in next.config.js file*

Let us look into an example of how you can use it with a real plugin. So let us check this one for example **https://github.com/vincent-herlemont/next-aws-lambda-webpack-plugin.**_Using this plugin you can use AWS Lambda functions as pages for your application. To implement this plugin you will need to install it first:

```
yarn add --dev next-aws-lambda-webpack-plugin

// or if you prefer NPM

npm install --save-dev next-aws-lambda-webpack-plugin
```

***Figure 1.13**: Command to install plugin into your project*

And then change the WebPack configuration like this to enable it:

```
const nextConfig = {
  reactStrictMode: true,
  styledComponents: true,
   webpack: (config, nextConfig) => {
        const newConfiguration = config.plugins.push(new GenerateAwsLambda(nextConfig))
        return newConfiguration
    },
  env: {
    SECRET: process.env.SECRET
  }
}

module.exports = nextConfig
```

***Figure 1.14**: Code in next.config.js file*

How to use TypeScript in NextJS

In the last NextJS version, there is no need for a special configuration of WebPack for using **TypeScript**

What if I have a NextJS project that was created with JS only?

If you have a NextJS project that was created without typescript just add a **`tsconfig.json`** file into your root and rerun your development server with **`yarn run dev`** (**or `npm run dev`**) after that NextJS will show you the next steps to proceed. You will need to enter these commands to fully configure your project

```
// With Yarn
yarn add --dev typescript @types/react @types/node

// With NPM
npm install --save-dev typescript @types/react @types/node
```

***Figure 1.15**: Commands to add typescript into the project*

How to use SCSS in NextJS

In a modern version of NextJS, you do not need to have a special configuration for using SCSS. Just rename your CSS files to SCSS and it will be automatically built. No matter how you created the application we highly recommend storing your style files in a separate folder.

Please note that using the style files should have a naming convention that requires a style name to be like this **`<your-style-name>.module.scss`**. That is a requirement because all global styles should be declared in your **`_app.ts`** file and connected to the application. But if you have custom styles for your different pages, please name your styles with module after the file name. It is a part of the framework and can't be reconfigured

How to enable and use styled components

In the modern React world, we can reuse components that were created for different proposals and frameworks. Using the **Styled Components** plugin will enable the feature to reuse components from web React to mobile React Native and from mobile to NextJS framework.

To enable this plugin to enter these commands in your CLI:

```
// With Yarn
yarn add styled-components

// With NPM
npm install --save styled-components

// Do not forget to add these types using @types
npm install --save @types/styled-components
```

***Figure 1.16**: Command to add Styled Components into project*

After that you will need to add a new configuration in your **next.config.js**:

```
const nextConfig = {
  reactStrictMode: true,
  styledComponents: true,
  env: {
    SECRET: process.env.SECRET
  }
}

module.exports = nextConfig
```

***Figure 1.17**: Code in next.config.js file*

Now we can use the Styled Components feature in our project as in the code below:

```
import type { NextPage } from 'next';
import { createGlobalStyle } from 'styled-components';

const GlobalStyle = createGlobalStyle`
    body {
        background-color: #ebebeb;
        min-height: 100vh;
        padding: 0.5rem;
        margin-top: 0;
        font-family: Verdana;
    }
`;

const StyledPage: NextPage = () => {
  return (
    <>
        <GlobalStyle />
        <div>This page will be styled with global
style</div>
  )
};

export default StyledPage;
```

***Figure 1.18**: Code for example styled page with Styled Component inside*

How to create a multipage app

As you remember the main element in the NextJS structure is a page. The page is a ReactJS component file (**`*.jsx`** or in our case **`*.tsx`**) that contains the code for a separated single page. The magic of NextJS starts in a place where you do not need to initialize or configure a possibility to create more than one page in the project. Each file that will be placed in the **pages** folder will be automatically wrapped with a link and could be called by URL request. So your file structure should look like this

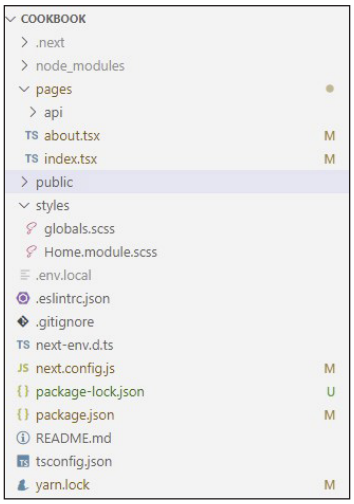


***Figure 1.19**: The project folder structure*

Here you can see we have two pages in our project. Let us put some code inside them and create a link between them:

```
import type { NextPage } from 'next'
import Link from 'next/link'
import styles from '../styles/Home.module.scss'

const Home: NextPage = () => {
  return (
    <div className={styles.container}>
      <h1>Hello there ! This is the main page of CookBook</h1>
      <Link href="/about"><a className={styles.link}>About</a></Link>
    </div>
  )
}

export default Home
```

***Figure 1.20**: Code from home page component*

And the second page with pretty similar code inside

```
import type { NextPage } from 'next'
import Link from 'next/link'
import styles from '../styles/About.module.scss'

const About: NextPage = () => {
  return (
    <div className={styles.container}>
      <h1>Hello there ! This is the About page of CookBook</h1>
      <Link href="/"><a className={styles.link}>Main</a></Link>
    </div>
  )
}

export default About
```

***Figure 1.21**: Code from second page component*

Now when we run the development server, we will see the page like this:

***Figure 1.22**: Result view for the home page*

And by the click on the **`About`** link we will open the page with different content like this:

***Figure 1.23**: Result view for the second page*

As we can see, both the pages are connected by the link automatically and no configuration is required for simple routing like this.

Also, you can define sub-folders inside your **`pages`** folder. That will also create a link to your page as well. For example, you can create **`pages/articles/folders`** and organize

its single page application there using **index.tsx** as a root file for the application and any name for other pages. The routing rules will be the same without dependency to the level of deep.

How to change pages - Routing tools

Even having such powerful tools for page creation as we mentioned before in the real-world application examples that feature still will be not enough. Pre-defined pages could be difficult to create and support. Also, we will need to have a big amount of file duplicates that are also not a good example of application architecture.

To solve this issue NextJS provides a useful mechanism with predefined paths. For example, we need to add more recipes to our CookBook application and the routes for them will look like this pattern: **https://cook.book/recipes/HealthyBreakfast**. In NextJS we use a dynamic route and it will expect the same file structure as in the pattern. But in our case, the last part should be dynamic so we can't just name the file **HealthyBreakfast**, so to solve it the file structure should look like this:

***Figure 1.24**: Project folder structure after file creation*

Now we can create the code for a file named **`[recipe].tsx`**. The brackets in the name are required by the framework to mark the file as the dynamic route. Please put this code into the file:

```
import { useRouter } from 'next/router'

const Recipe = () => {
  const router = useRouter()
  const { recipe } = router.query

  return <p>Recipe: {recipe}</p>
}

export default Recipe
```

***Figure. 1.25**: Code from [recipe].tsx file*

Now when we reload the page in the required URL, we will see that result on our screen:

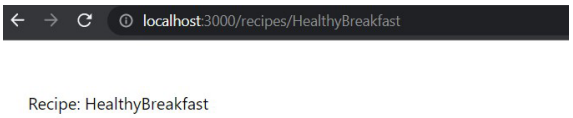


***Figure 1.26**: Result of [recipe].tsx rendering*

Nice! Now we can create any amount of recipe pages that could be ever required. In our code, we use the hook that is called **`useRouter`.** That hook will collect whole information about the router that we could need for our application. In the current case, we are getting query information data. Also, if we will need to get more data like in this example: **http://localhost:3000/recipes/HealthyBreakfast?additionalData=sugar-not-included ,**

we can get the information that is the in **`additionalData`** variable by changing the router query code like this:

```
const { recipe, additionalData } = router.query
```

***Figure 1.27**: Router query*

After that, we could use this variable inside the code as it could be required.

Anyway, we can also name folders as dynamic folders if we would need them. For example, our CookBook will start growing and we will need to group our recipes into groups like Breakfast, Dinner, and Soups. Then the URL should be done like this **http://localhost:3000/recipes/breakfast/HealthyBreakfast.** To solve it we can just create 3 folders and put the dynamic files in each one. But The page design will be the same so we do not need such a complicated structure the best solution will be to create a dynamic folder structure:

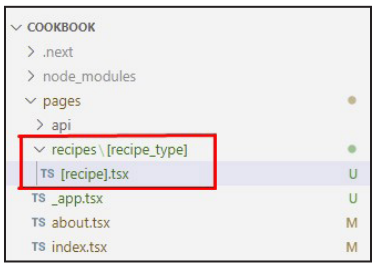


***Figure 1.28**: Project folder structure*

And in the end, the code of the page will look like this to react to dynamic route changes

```
import { useRouter } from 'next/router'

const Recipe = () => {
  const router = useRouter()
  const { recipe, recipe_type } = router.query


  return <p>Recipe: in <b>{recipe_type}</b> is <b>{recipe}</b></p>
}

export default Recipe
```

***Figure 1.29**: Code in recipe file*

You can also simplify the structure if there is a possibility to not think about how complex the parameters query is. For example, **http://localhost:3000/recipes/breakfast/sugar-free/HealthyBreakfast.** And it could grow and grow and grow… So. In this case, we can get rid of sub-folders and create the file with the name **`[...recipes].tsx`** in a pages folder like this:

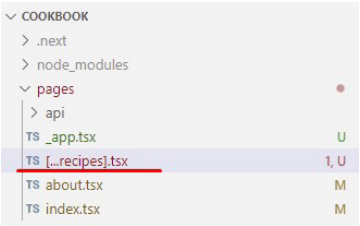


***Figure 1.30**: How to rename the file*

Also, we need to make changes to the page code. The main thing is that data for the router will come as an array so we will need to iterate it to show the information on the page.

```
import { useRouter } from 'next/router'

const Recipe = () => {
  const router = useRouter()
  const { recipes } = router.query

  return <div>Recipe: in
    {recipes && recipes.map((recipe: string, index: number) => {
        return <div key={index}><b>{recipe}</b></div>
    })}
  </div>
}

export default Recipe
```

***Figure 1.31**: New code in recipe file*

Pages that you create are wrapped with a special mechanism called Automatic Static Optimization. This feature contains both options of rendering that are possible for NextJS. If we use the static page option the route query will trigger after hydration. We will look closer at rendering options in the next chapters

How to change the router by the event?

We did a great job before and created the system that will give us an opportunity to create a multi-page application in a short timeline. But how to change the routes by the navigation click when we will have such big number of pages?

To do that lets create a link button in our code. That button will change the route to another page. For example, let us loop 2 pages with the '**next/previous'** button. To achieve it your code should look like this one:

```
import { useRouter } from 'next/router'

const Recipe = () => {
  const router = useRouter()
  const { recipes } = router.query
  let label = 'Next Recipe'
  let link = '/recipes/breakfast/HealthyBreakfast'

  if (recipes && recipes[2] === 'HealthyBreakfast') {
    label = 'Previous Recipe'
    link = '/recipes/breakfast/AnotherHealthyBreakfast'
  }

  return <div>
    Recipe: in
    {recipes && recipes.map((recipe: string, index: number) => {
        return <div key={index}><b>{recipe}</b></div>
    })}
    <button onClick={() => router.push(link)}>{label}</button>
  </div>
}


export default Recipe
```

***Figure 1.32**: Updated code of recipe file*

It is just an example. You can create your own expressions as you wish and generate the router link by the requirements.

How to change the page params state without running data fetching methods

Let us imagine that we have a page with recipes, and we have allowed our users to leave a comment on this page. After one or more years the number of comments became so huge that we were pushed to split all comments into pages that could get from API at once. The other problem is if we share the page or accidentally reload the page, we need to know the page number for the comments. If for the case where to store local data the solution can be easily solved - with sharing its more tricky part. NextJS routing module provides the functionality called **Shallow Routing** to solve this issue easily

So, we will:

1. Catch the comment age param on page load.
2. Update the number on the event. For the simple example, it will be the button:

```
import { useRouter } from 'next/router'
import { useEffect } from 'react'

const Recipe = ({initialData}) => {
  const router = useRouter()
  const { recipes, comment_page } = router.query
  let label = 'Next Recipe'
  let link = '/recipes/breakfast/HealthyBreakfast'
  let default_comment_pages_value = Number(comment_page) ?? 10
  if (recipes && recipes[2] === 'HealthyBreakfast') {
    label = 'Previous Recipe'
    link = '/recipes/breakfast/AnotherHealthyBreakfast'
  }

  useEffect(() => {
    // Init some default value
    router.push(link+'?comment_page='+default_comment_pages_value, undefined, { shallow: true })
  }, [])

  useEffect(() => {
      console.log(comment_page, initialData)
  }, [comment_page, initialData])

  const nextPageNumber = (page: string | string[] | undefined): number => {
    return Number(page) + 1
  }

  return <div>
    Recipe: in
    {recipes && recipes.map((recipe: string, index: number) => {
        return <div key={index}><b>{recipe}</b></div>
    })}
    <button onClick={() => router.push(link)}>{label}</button>
    <button onClick={() => router.push(link+'?comment_page='+nextPageNumber(comment_page), undefined, { shallow:
true })}>Change comment page</button>
  </div>
}

Recipe.getInitialProps = () => {
  const initialData = 'data on load: '+Math.random()
  return { initialData }
}

export default Recipe
```

***Figure 1.33**: Final version of recipe file*

As a result, we could change the variable parameter but the page init mechanism will fire only once because of the **shallow** parameter activated.

```
22:13:53.325 11 data on load: 0.0037369174468142585          [...recipes].tsx?af51:21
22:13:54.902 12 data on load: 0.0037369174468142585          [...recipes].tsx?af51:21
22:13:55.416 13 data on load: 0.0037369174468142585          [...recipes].tsx?af51:21
22:13:55.886 14 data on load: 0.0037369174468142585          [...recipes].tsx?af51:21
22:13:56.361 15 data on load: 0.0037369174468142585          [...recipes].tsx?af51:21
22:13:56.747 16 data on load: 0.0037369174468142585          [...recipes].tsx?af51:21
22:13:57.115 17 data on load: 0.0037369174468142585          [...recipes].tsx?af51:21
22:13:57.386 18 data on load: 0.0037369174468142585          [...recipes].tsx?af51:21
>
```

***Figure 1.34:** Console view after changes of recipe file*

As you can see the router param is changing but the initial data stays the same. That means that no fetching methods will be activated using **shallow** routing.

Conclusion

Still remembering the time when solving such problems could take several days. It was necessary to create the logic for changing the routing, process each request, receive data, create logic for the initial receipt of data, and so on. NextJS allows you to reduce the development time in solving such problems to several minutes. Thus, it is now possible to create a simple blog, provided that the serverless CMS is used. In the next chapters we will look at how to work with such data, but for now we will not go into this.

One should keep in mind that skill comes with practice. In this chapter, we looked at how to quickly deploy a project and create a simple application with a few pages. Practice. Create more pages and logic. Use ReactJS knowledge to create software. The design rules are the same.

In the next chapter we will start architect the application in modern way using design patterns and Test Driven development

Chapter 2
Using design patterns in NextJS

If everybody minded their own business, the world would go around a great deal faster than it does.

— Lewis Carroll

Introduction

In every business when we use, **NextJS** any part should take responsibility for its own scope, to not invent the wheel each time we can use good old programming patterns that will be able to solve a daily problem faster as keeping the codebase scalable and maintainable condition. The problems could be as short and solvable in many ways as complex and require more architectural decisions.

Speaking about the tools that we will use in this chapter - we are going to use design patterns in the app-building process. As you know there are several rules and requirements in the application-creating process and in the code quality requirements also.

In this book, we will use only three of them just to keep in the topic on **NextJS** but you can check and use other patterns in real life. The name of the patterns is Singleton, Builder, and Strategy. These patterns are mostly used in any scale of application from the beginners' small to really big and scaled applications.

Structure

- How to optimize your SPA and Router with patterns?
  - Writing Singleton pattern for data objects
  - Writing Builder pattern to operate the data
  - Writing strategy pattern for page-changing intent
- Using Test-Driven Development for safety and management
  - How to configure the TDD environment?
  - Writing your first component in a test-first way
- Conclusion

Objectives

In this chapter, we will learn how to optimize our core with simple design patterns that will give us the possibility to easily scale and refactor the application in any project timeline. Also in the second part of the chapter, we will introduce the TDD way of developing the NextJS application to make our code more safe and manageable for the project team members.

Optimizing your SPA and router with patterns

Let us deep down into a rabbit hole to figure out what problems in SPA and by routing we can solve using design patterns and if it is a real problem at all.

In the real world, the application can contain logic that is used in more than one place, and this logic can also be complex algorithms that require to use of more than one function or class. For such cases application development introduces the design patterns. The most advantage of using design patterns is that you do not need to invent a logic or algorithm but use the most well-known scheme. It will give less complexity to test and debug applications.

Writing Singleton pattern for data objects

To understand what is Singleton let`s imagine that to create the application we would require to get a new laptop each time we want to create a new file. Sounds complicated and also we will have too many laptops that are used for the same task. As the result, we will overflow our working space with laptops. The same will

happen with the data objects if we will create a new instance each time we want to work with the data that already exists.

In a big scaled app, we will need to use classes and objects that will be created in several places of our code. That could create overuse of memory for the browser. Of course, we live in a world with super powerful processors and do not have an efficient memory but, that does not mean that we can spend it all the time. In JavaScript, we, have to understand that resources using not only for calculating and working with the data but also for the UI and animation. To make your application work smooth and predictable we always need to think about optimization.

For such cases, we can use a Singleton pattern that will help us to solve several problems. First of all, we can be sure that a class is created only once that will save our resources. On the other hand, we can control the global access to the instance and use it to share the data between other instances.

The limitation of using this pattern can be that it is solving more than one problem as a class so it will violate the single responsibility principle.

This is a good practice to use in data fetching as we can solve many problems with data sharing and caching data only with only wrapping our logic with this pattern.

The Objective Oriented world will appear like *Figure 2.1:*

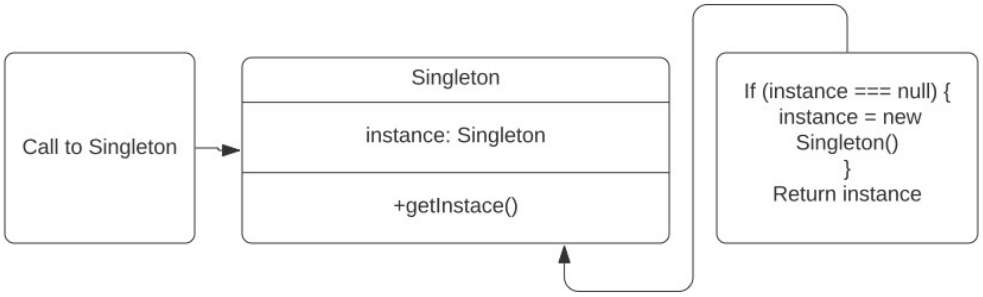


Figure 2.1: *Singleton schematic object view*

Where will we use this pattern in our application? We will use it only for data that we will share between components.

Let us try to create one object as a Singleton. To do it we will create the folder named "core" in the folder **"./pages"** to have not even shared elements between components but have it shared even with API. Then create a file named **"data_service.ts".** That will be our first shared object that will contain some information inside.

Take a look at the *Figure 2.2* with a code example. That code we will add to the file that we just created.

```
class ArticleDataService {
    private static instance: ArticleDataService;
    private constructor() { }

    public static getInstance(): ArticleDataService {
        if (!ArticleDataService.instance) {
            ArticleDataService.instance = new ArticleDataService();
        }

        return ArticleDataService.instance;
    }

}
```

Figure 2.2: *Singleton class instance*

The logic of the code is pretty simple. We are creating an instance of the class on creating only in case it is not created. In case it is already created we will use the current instance.

Please note! There is no silver bullet in using patterns as Singleton. Be careful in choosing. I recommend using it only for the data modeling for the application

Now we can do some architecture to figure out what data structure we will have to share data between components.

As we are playing around with the blogging system let us use some data for the articles so we need to collect the data about the page into properties of the class. In the real-world app, we can use as many such services as we want. We will start from one in our basic example here.

For the article, we will require several fields to have:

- **Title**: String
- **Author**: String
- **Content**: String
- **Category**: Object
- **Image**: String (it will be an URL)
- **Creation date**: Date

- **Allow comments flag**: Boolean
- **Status**: Enum

The code of the class for these requirements will look like the following figure:

```
interface ICategory {
  title: string
  url: string
}

interface IArticle {
    id: number
    title: string
    author: string
    content: string
    category: ICategory
    image: string
    createdAt: string
    allowComments: boolean
    status: 'public' | 'draft'
}

class ArticleDataService {
    private static instance: ArticleDataService;
    private constructor() { }
    private currentArticle: IArticle

    public static getInstance(): ArticleDataService {
        if (!ArticleDataService.instance) {
            ArticleDataService.instance = new ArticleDataService();
        }
        return ArticleDataService.instance;
    }

    getArcticle(id: number) {
        /**
         * Here we will add the currentArticle initiation later
         */
    }

}
```

Figure 2.3: *Article Data class using interfaces and Singleton pattern*

Now we can re-use and call the data of the article from any place in our code and it will be shared between the modules. How it will help us if technically we can just send all data as props into all components on the page? So, for this exact example, we do not need the data to be props as we do not plan to mutate the data while rendering and using the page. It will be rendered once and never changed until the whole page is reloaded. Also, we will not need to change the props of the component

in case if the data structure is changed so we will be double-safe in this case. That does not mean that we will get rid of using props in our architecture. It is just a possibility to use data in different places.

As we will use the API functionality of NextJS in future chapters here we will just create some mock data to use in the app. Please create some JSON files in the folder "mocks" in the "pages" folder.

The file with mock data will contain this data as shown in *Figure 2.4*:

```
[
    {
        "id": 1,
        "title": "Breakfast burger",
        "author": "John Wik",
        "content": "Some content for the blog",
        "category": {
            "title": "Breakfast food",
            "url": "/categories/breakfast"
        },
        "image": "shorturl.at/hluIO",
        "createdAt": "2022-05-01",
        "allowComments": true,
        "status": "public"
    }
]
```

Figure 2.4: *The mock-data JSON that will be used in the app*

For the test, you can create more than one element in the array of data. To get the data we will create the **`index.ts`** file in the "pages/mocks" folder.

This file will contain a function that will imitate API calls and return the data as the following figure:

```
import articles from "./articles.json"

const getMock = {
    articles
}

export { getMock }
```

Figure 2.5: *Aggregator object for the mock data*

After that, we can implement this function into our Singleton class to get the data ones it is required.

Let us do some updates in the service class to implement mock data receive: (*Figure 2.6*):

```
class ArticleDataService {
    private static instance: ArticleDataService;
    private constructor() { }
    private articles: {[key:string]: IArticle} = {}
    private navigation: Array<string> = []

    public static getInstance(): ArticleDataService {
        if (!ArticleDataService.instance) {
            console.log('ArticleDataService new instance')
            ArticleDataService.instance = new ArticleDataService();
        }
        return ArticleDataService.instance;
    }

    getNavigation() {
        if (Array.isArray(this.navigation) && this.navigation.length === 0) {
            this.navigation = getMock.articles.map(atricle => atricle.id)
        }
        return this.navigation
    }

    getArcticle(id: string) {
        if (!this.articles[id]) {
            this.articles[id] = getMock.articles.find( (article) => article.id === id ) as
IArticle}
        return this.articles[id]
    }

}
```

Figure 2.6: *Updated service with managing data methods*

Leaving the **`console.log`** statement to show you how it will work. The log statement will trigger only once even if you will do actions on the page and change the router state. The reload of the class will happen only if we do a hard page reload.

As you probably can imagine - this way of service use might be used instead of using Redux or any state management system. But please do not confuse patterns. Redux is a specific implementation of the Flux pattern that provides a more streamlined and efficient way of managing the application state. So using Redux we can subscribe and know when the state was changed and what was before the changes. Singleton is a good thing only in case we need to share data or cache the data.

The data is loaded and we will need to operate with this data somehow. For these requirements, we can use another pattern called Builder.

Writing builder pattern to operate the data

Now when we have a singleton for data we can proceed with page creation. In simple examples, there is no issue with just getting the data and then putting it into the template. Let us scale the example from a basic "`Hello-world`" application to something more specific and enterprise.

In this case, we will need to operate this data into something special before we do render. In this case, we can use another pattern called Builder.

For example, we do a Food blog that contains recipes for baking Burgers. There are hundreds of different burgers but the steps of creation are quite same. We will take a basic example that will contain only a few of them as it could be very complex in the end. Let us take an example where exists only these steps:

1. Baking top buns part with seeds.
2. Baking top buns part without seeds.
3. Baking meat.
4. Baking fish.
5. Baking chicken.
6. Prepare burger sauce.
7. Prepare fish sauce.
8. Prepare special sauce.
9. Baking bottom buns part.
10. Grill burger with cheese.

Most of our burgers will have these steps to produce a burger. But not all the burgers will have all the steps.

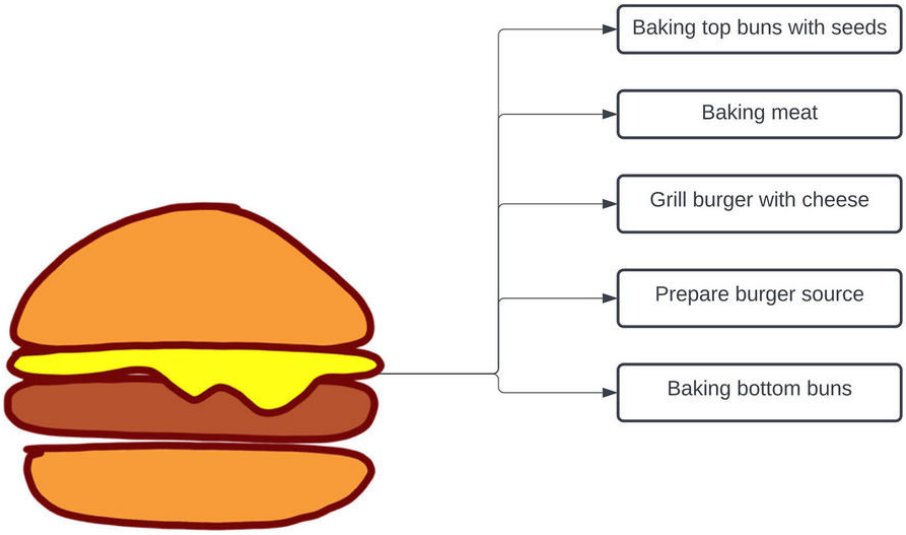


***Figure 2.7**: Schematics for the burger producing steps*

In *Figure 2.7* we can see an example of what the building process will look like for some simple burgers with meat. We will use only half of the possible steps to build some products.

This is a real-world example of how the Builder pattern is working. Now we can use this knowledge in the programming language world.

Let us try to write some code for this pattern. For typescript it will look like this:

```
interface IBuilder {
    backingTopBunsPartWithSeeds(): void;
    backingTopBunsPartWithoutSeeds(): void;
    backingMeat(): void;
    backingFish(): void;
    backingChicken(): void;
    prepareBurgerSouce(): void;
    prepareFishSouce(): void;
    prepareSpecialSouce(): void;
    backingBottomBunsPart(): void;
    grillBurgerWithCheese(): void;
}
```

***Figure 2.8:** Builder interface for the implementation*

The convention of using an "I" prefix for interfaces was popularized by Microsoft in their .NET framework, where it is a widely used convention for naming interfaces. This convention has since been adopted by many other programming communities, including the TypeScript community

This interface will be a main part of the builder class as it contains all methods that will be used in data creation.

Now having such an interface, we can create the builder class that will appear like *Figure 2.9*:

```
class BurgerBuilder implements IBuilder {
    backingTopBunsPartWithSeeds(): void {
        console.log('Top bun with seeds is builded')
    }
    backingTopBunsPartWithoutSeeds(): void {
        console.log('Top bun without seeds is builded')
    }
    backingMeat(): void {
        console.log('Meat is builded')
    }
    backingFish(): void {
        console.log('Fish is builded')
    }
    backingChicken(): void {
        console.log('Chicken is builded')
    }
    prepareBurgerSouce(): void {
        console.log('Burger souce is builded')
    }
    prepareFishSouce(): void {
        console.log('Fish souce is builded')
    }
    prepareSpecialSouce(): void {
        console.log('Special souce is builded')
    }
    backingBottomBunsPart(): void {
        console.log('Bottom bun is builded')
    }
    grillBurgerWithCheese(): void {
        console.log('Grill with cheese is builded')
    }
}
```

Figure 2.9 *Builder class realization using builder interface*

After that, we will need a director class that can contain all options on our burger menu. Note that the Director class is not necessary to exist but it will be more simple to call one class that contains the menu of the burgers for our example. This class will look like this:

```
import { IBuilder } from "./burger-builder";

class BurgerDirector {
    private builder!: IBuilder

    constructor(builder: IBuilder) {
        this.setBuilder(builder);
    }

    public setBuilder(builder: IBuilder): void {
        this.builder = builder
    }

    public buildHamburger(): void{
        this.builder.backingTopBunsPartWithSeeds()
        this.builder.backingMeat()
        this.builder.grillBurgerWithCheese()
        this.builder.prepareBurgerSouce()
        this.builder.backingBottomBunsPart()
    }

}

export { BurgerDirector }
```

***Figure 2.10**: Builder director class to orchestrate builder*

To activate the building we need to add the creation of a builder and director to our component with the following code:

```
const burgerBuilder = new BurgerBuilder()
const burgerDirector = new
BurgerDirector(burgerBuilder)
```

***Figure 2.11:** Activation of building using all instances that we created*

We can place this code in any part of the code but we highly recommend doing it in the components part before the rendering part. After that in the place where we get the data we can use the builder like *Figure 2.12:*

```
  useEffect(() => {
    if (pid) {
      setContent( {...ArticleDataService.getInstance().getArticle(recipes[1])}
)     burgerDirector.buildHamburger();
    }
  }, [recipes, burgerDirector]);
```

Figure 2.12: *Build hamburger on page load using useEffect hook*

Now we can see that the burger is perfectly built if we open the console as illustrated in *Figure 2.13:*

```
Top bun with seeds is builded          burger-builder.ts?13c6:16
Meat is builded                        burger-builder.ts?13c6:22
Grill with cheese is builded           burger-builder.ts?13c6:43
Burger souce is builded                burger-builder.ts?13c6:31
Bottom bun is builded                  burger-builder.ts?13c6:40
```

Figure 2.13: *Console log result after page load*

You can use any options to build the burger but this is how it works for any case that we will use in this book.

This is an abstract example. But to use it in the real-world application we can use the previous experience of the Singleton pattern and add some data inside.

To achieve it we will need to add some code to our project. The defined steps of what we will do will look like this:

1. We will require some mock data that will be created as we did it before for the articles.
2. We will need a singleton service class to operate with the data.
3. We will define a recipe as a product and make the builder return the product as a result of the build process.
4. We will define steps as an enum.
5. We will define logic regarding enums.

The complete code after these requirements will look like this. The mock data as depicted in *Figure 2.14 :*

```
{
    "backingTopBunsPartWithSeeds": "We are baking the top bun with seeds",
    "backingTopBunsPartWithoutSeeds": "We are baking the top bun without seeds",
    "backingMeat": "We are backing the meat",
    "backingFish": "We are backing the fish",
    "backingChicken": "We are backing the chicken",
    "prepareBurgerSouce": "We are preparing the burger souce",
    "prepareFishSouce": "We are preparing the fish souce",
    "prepareSpecialSouce": "We are preparing the special souce",
    "backingBottomBunsPart": "We are backing the bottom bun",
    "grillBurgerWithCheese": "We are grill burger wth cheese"
}
```

Figure 2.14: *Mock-date for the builder*

After that, we will create the file **"burger-config.ts"** that will contain an enum with the required keys for this data. That enum will help us to define all names that we will use in the builder as each key will contain the data key and the name of the method that will be used to call for this data. Having this enum we will define the connection between data and code, and we will also solve the future issue with naming convection of methods names.

Check the next *Figure 2.15* to find out what the code will look like:

```
enum BurgerSteps {
    TOP_BUNS_WITH_SEEDS = "backingTopBunsPartWithSeeds",
    TOP_BUNS_WITHOUT_SEEDS = "backingTopBunsPartWithoutSeeds",
    MEAT = "backingMeat",
    FISH = "backingFish",
    CHICKEN = "backingChicken",
    BURGER_SOUCE = "prepareBurgerSouce",
    FISH_SOUCE = "prepareFishSouce",
    SPECIAL_SOUCE = "prepareSpecialSouce",
    BOTTOM_BUNS = "backingBottomBunsPart",
    GRILL_BURGER_WITH_CHEESE = "grillBurgerWithCheese"
}

export { BurgerSteps }
```

Figure 2.15: *Enum for the Builder*

The data service will require a bit more logic rather than just getting the data and providing the data. As we are making the system that will be ready for real-world API calls we will have the method to get each step of the recipe by the key. We will cache this data inside the singleton to reuse it in the next creation of the burger. To complete this issue we will need this class like in the following figure:

```
import { getMock } from "../mocks"
import { BurgerSteps } from "./burger-config";

class BurgerDataService {
    private static instance: BurgerDataService;
    private steps!: {key: BurgerSteps, value: string}[]
    private constructor() {
        this.steps = []
    }

    public static getInstance(): BurgerDataService {
        if (!BurgerDataService.instance) {
            console.log('BurgerDataService new instance')
            BurgerDataService.instance = new BurgerDataService();
        }
        return BurgerDataService.instance;
    }

    getStep(key: BurgerSteps) {
        let currentStep = this.steps.find( (step: {key: BurgerSteps, value: string}) => step.key === key
)       if (!currentStep) {
            currentStep = {
                key,
                value: getMock.baking[key]
            }
            this.steps.push(currentStep)
        }
        return currentStep
    }

}

export { BurgerDataService }
```

***Figure 2.16**: Service to manage burger data*

As you can see, we have the "**`getStep`**" method to get the step by the key name that we described before in the BurgerSteps enum and for each method call, we will collect the steps array, which will be reused if we will need it, instead of the API call. Why do need to optimize this at all? Why just not have a call each time we need the information? The answer is simple - Money. In the modern architectures that are used for the deployment of the applications the cloud solutions mostly cost

per amount of calls that we are making during application use. In this case, if we will have fewer bills at the end of the month. Also we will increase the speed of the application as we do not need to wait for the server to get the information that already exists in the application

Finally, we will do some changes in the builder and director classes to fit the new requirements regarding having the enum in the code: (*Figure 2.17*)

```
interface IBuilder {
    [BurgerSteps.TOP_BUNS_WITH_SEEDS]: () => void
    [BurgerSteps.TOP_BUNS_WITHOUT_SEEDS]: () => void
    [BurgerSteps.MEAT]: () => void
    [BurgerSteps.FISH]: () => void
    [BurgerSteps.CHICKEN]: () => void
    [BurgerSteps.BURGER_SOUCE]: () => void
    [BurgerSteps.FISH_SOUCE]: () => void
    [BurgerSteps.SPECIAL_SOUCE]: () => void
    [BurgerSteps.BOTTOM_BUNS]: () => void
    [BurgerSteps.GRILL_BURGER_WITH_CHEESE]: () => void
    getRecipe(): Array<string>
    resetBuilder(): void
}
```

Figure 2.17: *Builder interface*

As you see we can use the enum element name as a name for any method or property.

Using this possibility, we can rename and rewrite all required methods for the interface using the new enum element.

We will add two more methods that will add logic that will allow us to return the recipe as a product and reset the builder to have the ability to start other another burger in our process lanes as shown in *Figure 2.18:*

```
class BurgerBuilder implements IBuilder {
    private recipe: Array<string> = []
    getRecipe(): Array<string> {
        const steps = this.recipe
        this.resetBuilder()
        return steps
    }
    resetBuilder() {
        this.recipe = []
    }
    setStep(name: BurgerSteps) {
        this.recipe.push(
            BurgerDataService
                .getInstance()
                .getStep(name).value
        )
    }
    [BurgerSteps.TOP_BUNS_WITH_SEEDS](): void {
        this.setStep(BurgerSteps.TOP_BUNS_WITH_SEEDS)
        console.log('Top bun with seeds is builded')
    }
    [BurgerSteps.TOP_BUNS_WITHOUT_SEEDS](): void {
        this.setStep(BurgerSteps.TOP_BUNS_WITHOUT_SEEDS)
        console.log('Top bun without seeds is builded')
    }
    [BurgerSteps.MEAT](): void {
        this.setStep(BurgerSteps.MEAT)
        console.log('Meat is builded')
    }
    [BurgerSteps.FISH](): void {
        this.setStep(BurgerSteps.FISH)
        console.log('Fish is builded')
    }
    [BurgerSteps.CHICKEN](): void {
        this.setStep(BurgerSteps.CHICKEN)
        console.log('Chicken is builded')
    }
    [BurgerSteps.BURGER_SOUCE](): void {
        this.setStep(BurgerSteps.BURGER_SOUCE)
        console.log('Burger souce is builded')
    }
    [BurgerSteps.FISH_SOUCE](): void {
        this.setStep(BurgerSteps.FISH_SOUCE)
        console.log('Fish souce is builded')
    }
    [BurgerSteps.SPECIAL_SOUCE](): void {
        this.setStep(BurgerSteps.SPECIAL_SOUCE)
        console.log('Special souce is builded')
    }
    [BurgerSteps.BOTTOM_BUNS](): void {
        this.setStep(BurgerSteps.BOTTOM_BUNS)
        console.log('Bottom bun is builded')
    }
    [BurgerSteps.GRILL_BURGER_WITH_CHEESE](): void {
        this.setStep(BurgerSteps.GRILL_BURGER_WITH_CHEESE)
        console.log('Grill with cheese is builded')
    }
}
```

Figure 2.18: *Builder realization using the interface*

Also, we need to add some refactoring to director-class as shown in *Figure 2.19:*

```
class BurgerDirector {
    private builder!: IBuilder

    public setBuilder(builder: IBuilder): void {
        this.builder = builder
    }

    public buildHamburger(): void{
        this.builder[BurgerSteps.TOP_BUNS_WITH_SEEDS]()
        this.builder[BurgerSteps.MEAT]()
        this.builder[BurgerSteps.GRILL_BURGER_WITH_CHEESE]()
        this.builder[BurgerSteps.BURGER_SOUCE]()
        this.builder[BurgerSteps.BOTTOM_BUNS]()
    }

}
```

***Figure 2.19**: Builder director to orchestrate builder*

Finally, we have the whole burger building process in the code and we can build any burger recipe for an article in our food blog. We can add, change or remove steps as many times as we want and it will not take a lot of refactoring as even the names of the methods are centralized in the enum.

Figure 2.20 explained the call of the builder to build a burger. Add a **`console.log`** function to check what is in the product now:

```
  useEffect(() => {
    if (pid) {
      setContent( {...ArticleDataService.getInstance().getArticle(recipes[1])}
)     burgerDirector.buildHamburger();
      console.log("getting the recipe", burgerBuilder.getRecipe());
    }
  }, [recipes, burgerDirector]);
```

***Figure 2.20** Initiate the builder on page load using the useEffect hook*

The result in the console will look like this: (*Figure 2.21*)

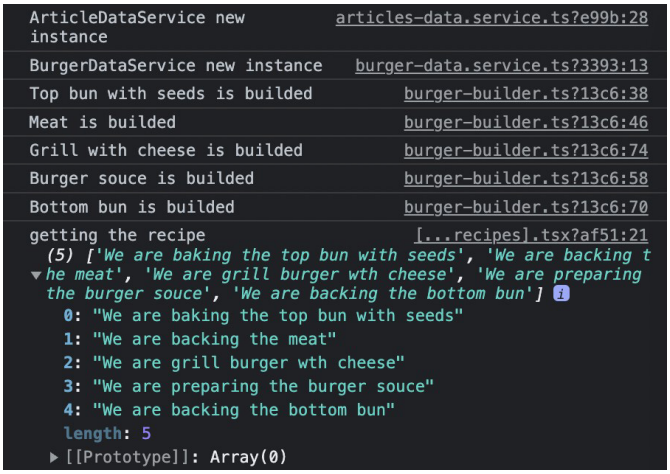


Figure 2.21: *Console log result after the page load*

The burger is done and we can move to the next pattern that called Strategy.

Writing Strategy pattern for page changing intent

Speaking about burgers - there are several of them and each one will require its personal way of backing. In the other words, we will use different strategies to bake different burgers. This issue is possible to solve using a regular "if" expression, but it will be more beautiful to use a Strategy pattern here.

The pattern basis is that we can take similar algorithms and group them into their scoped classes and then switch these classes while the application is working. Having this feature we can change the recipe in real-time having different strategy classes that are highly independent of one another.

For the current example we will make it work using the following steps:

1. We will create Strategy classes to use them as the action object instead of expressions.
2. We will use a naming convention with an Enum that will contain the possible method names inside.
3. We will create the class with business logic where we will create the method that will initiate chosen strategy class as a parameter.

The working scheme of the pattern is described in the *Figure 2.22*:

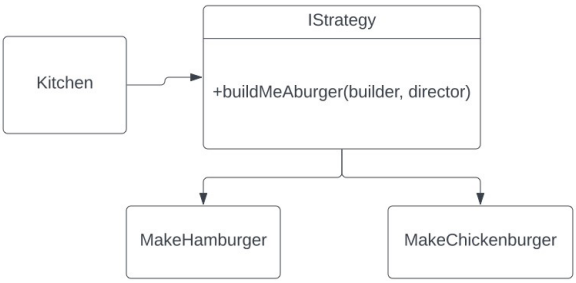


Figure 2.22: *Strategy object schematics*

We have enough theory here so we can start coding after having all requirements. In the first place we will need the burger type in our article mock data and add the burger type parameter inside like this: (*Figure 2.23*):

```
{
        "id": "breakfast_burger",
        "title": "Breakfast burger",
        "author": "John Wik",
        "burger": "hamburger",
        "content": "...",
        "category": {
            "title": "Breakfast food",
            "url": "/categories/breakfast"
        },
        "image": "shorturl.at/hluIO",
        "createdAt": "2022-05-01",
        "allowComments": true,
        "status": "public"
    },
```

Figure 2.23 *Updated mock data with burger type*

Then we will add some more configurations for the burgers. We will need enums to store the strategies for baking like this:

```
enum StrategiesNames {
    HAMBURGER = "hamburger",
    CHICKENBURGER = "chickenburger"
}

const Strategies = {
    [StrategiesNames.HAMBURGER]: new MakeHamburger(),
    [StrategiesNames.CHICKENBURGER]: new MakeChickenburger()
}
```

Figure 2.24*: Namespace for the strategies*

Now we have 2 baking strategies that will help us to make a Strategy pattern logic. Then having this we can create the interface and strategies classes: (*Figure 2.25)*

```
interface IStrategy {
    bakeMeAburger(burgerBuilder: BurgerBuilder, burgerDirector: BurgerDirector): Array<string>;
}

class MakeHamburger implements IStrategy {
    public bakeMeAburger(burgerBuilder: BurgerBuilder, burgerDirector: BurgerDirector): string[] {
        burgerDirector.buildHamburger()
        return burgerBuilder.getRecipe()
    }
}

class MakeChickenburger implements IStrategy {
    public bakeMeAburger(burgerBuilder: BurgerBuilder, burgerDirector: BurgerDirector): string[] {
        burgerDirector.buildChickenburger()
        return burgerBuilder.getRecipe()
    }
}
```

Figure 2.25: *Strategy classes that will be used in the application*

Now we can finally create the **`Kitchen class`** that will contain the logic of baking different burgers depending on what strategy has been chosen as shown in *Figure 2.26:*

```
class Kitchen {
    private strategy: IStrategy;

    constructor(strategy: IStrategy) {
        console.log('Strategy class is', strategy)
        this.strategy = strategy
    }

    public setStrategy(strategy: IStrategy) {
        console.log('strategy', strategy)
        this.strategy = strategy
    }

    public bakeSomething(burgerBuilder: BurgerBuilder, burgerDirector: BurgerDirector): void {
        console.log('Now Kitchen is on fire')
        const result = this.strategy.bakeMeAburger(burgerBuilder, burgerDirector)
        console.log('We baked:', result)
    }
}
```

Figure 2.26: *Context class with business logic for the application*

To make these updates work in the page component we will need also to update the code there to have the following code inside:

```
import { useRouter } from 'next/router'
import { useEffect, useMemo, useState } from 'react'
import { ArticleDataService, IArticle } from '../core/articles-data.service'
import styles from '../../styles/Recipes.module.scss'
import { BurgerDirector } from '../core/burger-director'
import { BurgerBuilder } from '../core/burger-builder'
import { IStrategy, Kitchen } from '../core/burger-strategy'
import { Strategies, StrategiesNames } from '../core/burger-config'

const Recipe = ({ initialData }: Partial<any>) => {
  const router = useRouter()
  const [content, setContent] = useState<IArticle>({} as IArticle);
  const { pid } = router.query
  const links = ArticleDataService.getInstance().getNavigation()
  const burgerBuilder = new BurgerBuilder()
  const burgerDirector = new BurgerDirector()
  burgerDirector.setBuilder(burgerBuilder)

  let burgerType = null
  let context: Kitchen
  let currentBurgerType: StrategiesNames
  const changeBurgerType = () => {
    console.log("current burger type", currentBurgerType)
    // Next line of code having no sence. We will just switch strategies to show the example is works
    const newStrategy = currentBurgerType === StrategiesNames.HAMBURGER ? StrategiesNames.CHICKENBURGER :
StrategiesNames.HAMBURGER
    currentBurgerType = newStrategy
    context.setStrategy(Strategies[currentBurgerType])
    context.bakeSomething(burgerBuilder, burgerDirector)
  }

  useEffect(() => {
    if (pid) {
      setContent( {...ArticleDataService.getInstance().getArticle(pid as string) } )
    }
  }, [pid])

  useMemo(() => {
    if(Object.keys(content).length > 0) {
      console.log("content.burger", content.burger)
      // we do not use this variables as state of the component as we do not need to see
      // changes of them in the template. Only the content variable will matters in this example
      burgerType = Strategies[content.burger as StrategiesNames]
      context = new Kitchen(burgerType)
      context.bakeSomething(burgerBuilder, burgerDirector)
    }
  }, [content])
```

Figure 2.27: *Initiation of strategies on page load*

Now on each reload we will bake a burger depending on what data will come from the article. We will have a possibility to change the strategy while using this page using the function **“changeBurgerType”.**

Starting this code will show us the same result as we had before but on calling the change burger type function it should look like this:

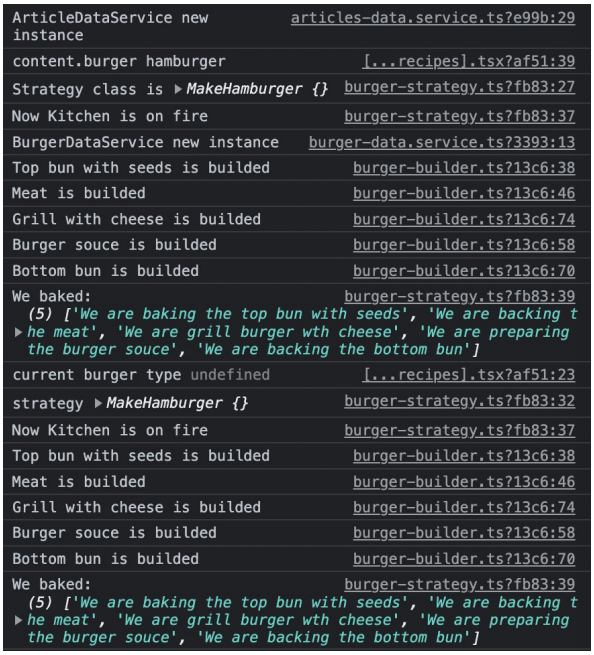


Figure 2.28: *Console log result after page reload*

As we see here we baked two different burgers while using one page.

I know that right now it looks like over-coding the simple issue. But please note that using patterns as using complex frameworks is not a good idea for the basic tasks. You should understand that your application is expected to be complex and scalable to use such a way of programming applications

Using test-driven development for safety and management

In the software development world, we cannot fully trust anybody, especially the code. The code can contain errors, bugs, and wrong logic. We can continuously check the application manually using requirements but as far as the application will grow we will lose control of the application quality. To avoid such situations, we should use the **Test-Driven Development** process (we will call it TDD to make it more short and precise). TDD is the way of creating applications that are based on short cycles of the development flow. In this process, we will not code the application but we will code the tests first before we create any code at all. That will mean that when the development is started, we will use requirements to create the tests and then create the code that will be created to pass these tests.

Generally, there is not always possible to create a maximal amount of tests before we made any application because in Agile (which is used in most application projects) we do not have strict requirements at the start. But the good news is that we do not need to write a big amount of tests. Any amount will be enough to follow this beautiful but not easy methodology of application creation.

Configuring the TDD environment

In this book, we will use several tools to create tests and make the application more development-safe. For the unit tests, we will need Jest and test library from React (as we use the react inside) and Playwright for the End-to-end tests. Having this our CI/CD will be in a safe place.

What is Jest? Jest is a Javascript testing library that will allow us to create unit tests that will be required before the application is rendered. In simple words, the library will help us to check the code safety before it will be rendered to production like this:

- The library will wrap the function with a call.
- The library will get the result of the function isolated from the application.
- The library will assert the result using expectations as a result of the job.

The next library that is required is React Testing library. This one is not necessary in the real world as we will use E2E tests. But, having a DOM test before rendering and having snapshots for each component will also add more safety to your application.

To add these libraries to your project type this in your console: (*Figure 2.29)*

```
npm install --save-dev jest babel-jest @testing-library/react @testing-library/jest-dom identity-obj-proxy react-
test-renderer
```

Figure 2.29: *Command to install required packages*

Use the flag **"`--legacy-peer-deps`"** for the command if you will see an error about the wrong version of the react.

Now we can configure Jest in our project. Add **"`jest.config.js`"** in the project root with this configuration as shown in *Figure 2.30:*

```
module.exports = {
  collectCoverageFrom: [
    '**/*.{js,jsx,ts,tsx}',
    '!**/*.d.ts',
    '!**/node_modules/**',
  ],
  moduleNameMapper: {
    /* Handle CSS imports (with CSS modules)
    https://jestjs.io/docs/webpack#mocking-css-modules */
    '^.+\\.module\\.(css|sass|scss)$': 'identity-obj-proxy',

    // Handle CSS imports (without CSS modules)
    '^.+\\.(css|sass|scss)$': '<rootDir>/__mocks__/styleMock.js',

    /* Handle image imports
    https://jestjs.io/docs/webpack#handling-static-assets */
    '^.+\\.(jpg|jpeg|png|gif|webp|avif|svg)$':
      '<rootDir>/__mocks__/fileMock.js',
  },
  testPathIgnorePatterns: ['<rootDir>/node_modules/', '<rootDir>/.next/'],
  testEnvironment: 'jsdom',
  transform: {
    /* Use babel-jest to transpile tests with the next/babel preset
    https://jestjs.io/docs/configuration#transform-objectstring-pathtotransformer--pathtotransformer-object */
    '^.+\\.(js|jsx|ts|tsx)$': ['babel-jest', { presets: ['next/babel'] }],
  },
  transformIgnorePatterns: [
    '/node_modules/',
    '^.+\\.module\\.(css|sass|scss)$',
  ],
}
```

Figure 2.30: *Configuration file to setup Jest*

Let us do some describing of what we created in the configuration:

- **`collectCoverageFrom`** is a filename pattern that will be used to reach out to the test files
- **`moduleNameMapper`** is a mapper for the style files with import inside
- **`testPathIgnorePatterns`** is a path pattern that will be ignored for test coverage
- **`testEnvironment`** is a rule for how we will test components. We use jsdom so do not forget to install it separately by typing "**`yarn add jest-environment-jsdom`**"
- **`transform`** is using a transpiler before we do coverage
- **`transformIgnorePatterns`** patterns to ignore for transpiler

If you face the problem with the text "Cannot find module 'react-dom/client'" just downgrade the test library in package.json like this "@testing-library/react": "12.1.5"

Now we can start creating our first test. To do it create the file **`index.test.jsx in __tests__ folder.`** In this file we will code the test like this:

```
import React from 'react'
import { render, screen } from '@testing-library/react'
import Home from '../pages/index.page'

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

    const heading = screen.getByRole('heading', {
      name: /Hello there ! This is the main page of CookBook/i,
    })

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

Figure 2.31: *Code for testing. We check that text is in component*

This basic test is rendering the home page and checking that it is having the heading element with text that is provided in the name section. Now we can run a test by typing "yarn test". The result will look like this:

```
yarn run v1.22.18
$ jest
 PASS  __tests__/index.test.jsx
  Home
    ✓ renders a heading (40 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.427 s, estimated 1 s
Ran all test suites.
✨  Done in 0.77s.
```

Figure 2.32: *Console log result after test start*

For now, you can just play around with Jest as we will proceed with the configuration of the test environment.

To init playwright in your project do this in your console in the root directory as shown in *Figure 2.33:*

```
npm init playwright
```

Figure 2.33: *Playwright installation command*

This will init the library in your root project and you will see this message in your console as shown in *Figure 2.34:*

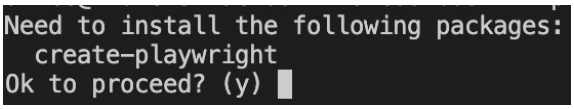


Figure 2.34: *Installation process for Playwright package*

Proceed with typing "**y**" in the console. When it asks you "Where to put your end-to-end tests? " type "e2e" to put your tests in the folder with this name. The library will create the example file with the most popular examples in this folder. You can play around to figure out how it works.

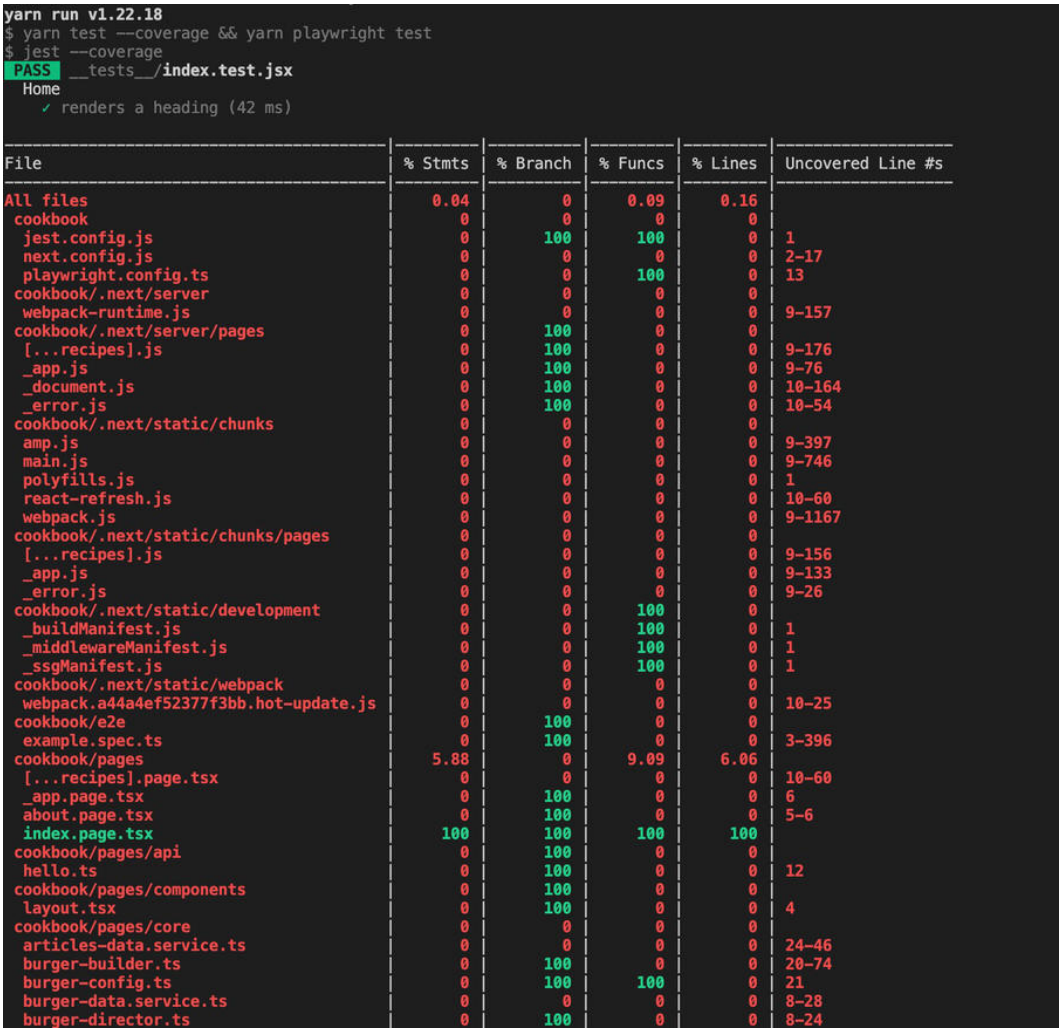
Now we have two test systems in the project and we need to avoid conflicts between them. To do it change your test ignore patterns in Jest configuration to the following figure:

Figure 2.35: *Configuration changes, required to proceed with test packages*

We can create the new task in **`package.json`** that will cover Jest and E2E tests. Add this into your package file as shown in *Figure 2.36:*

Figure 2.36: *Task that should be added to package.json file*

After that in your console, you will see the full report of our tests like the following *Figure 2.37*:



```
yarn run v1.22.18
$ yarn test --coverage && yarn playwright test
$ jest --coverage
 PASS  __tests__/index.test.jsx
  Home
    ✓ renders a heading (42 ms)

-----------------------------------------|---------|----------|---------|---------|-------------------
File                                     | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------------------------------|---------|----------|---------|---------|-------------------
All files                                |    0.04 |        0 |    0.09 |    0.16 |
 cookbook                                |       0 |        0 |       0 |       0 |
  jest.config.js                         |       0 |      100 |     100 |       0 | 1
  next.config.js                         |       0 |        0 |       0 |       0 | 2-17
  playwright.config.ts                   |       0 |        0 |     100 |       0 | 13
 cookbook/.next/server                   |       0 |        0 |       0 |       0 |
  webpack-runtime.js                     |       0 |        0 |       0 |       0 | 9-157
 cookbook/.next/server/pages             |       0 |      100 |       0 |       0 |
  [...recipes].js                        |       0 |      100 |       0 |       0 | 9-176
  _app.js                                |       0 |      100 |       0 |       0 | 9-76
  _document.js                           |       0 |      100 |       0 |       0 | 10-164
  _error.js                              |       0 |      100 |       0 |       0 | 10-54
 cookbook/.next/static/chunks            |       0 |        0 |       0 |       0 |
  amp.js                                 |       0 |        0 |       0 |       0 | 9-397
  main.js                                |       0 |        0 |       0 |       0 | 9-746
  polyfills.js                           |       0 |        0 |       0 |       0 | 1
  react-refresh.js                       |       0 |        0 |       0 |       0 | 10-60
  webpack.js                             |       0 |        0 |       0 |       0 | 9-1167
 cookbook/.next/static/chunks/pages      |       0 |        0 |       0 |       0 |
  [...recipes].js                        |       0 |        0 |       0 |       0 | 9-156
  _app.js                                |       0 |        0 |       0 |       0 | 9-133
  _error.js                              |       0 |        0 |       0 |       0 | 9-26
 cookbook/.next/static/development       |       0 |        0 |     100 |       0 |
  _buildManifest.js                      |       0 |        0 |     100 |       0 | 1
  _middlewareManifest.js                 |       0 |        0 |     100 |       0 | 1
  _ssgManifest.js                        |       0 |        0 |     100 |       0 | 1
 cookbook/.next/static/webpack           |       0 |        0 |       0 |       0 |
  webpack.a44a4ef52377f3bb.hot-update.js |       0 |        0 |       0 |       0 | 10-25
 cookbook/e2e                            |       0 |      100 |       0 |       0 |
  example.spec.ts                        |       0 |      100 |       0 |       0 | 3-396
 cookbook/pages                          |    5.88 |        0 |    9.09 |    6.06 |
  [...recipes].page.tsx                  |       0 |        0 |       0 |       0 | 10-60
  _app.page.tsx                          |       0 |      100 |       0 |       0 | 6
  about.page.tsx                         |       0 |      100 |       0 |       0 | 5-6
  index.page.tsx                         |     100 |      100 |     100 |     100 |
 cookbook/pages/api                      |       0 |      100 |       0 |       0 |
  hello.ts                               |       0 |      100 |       0 |       0 | 12
 cookbook/pages/components               |       0 |      100 |       0 |       0 |
  layout.tsx                             |       0 |      100 |       0 |       0 | 4
 cookbook/pages/core                     |       0 |        0 |       0 |       0 |
  articles-data.service.ts               |       0 |        0 |       0 |       0 | 24-46
  burger-builder.ts                      |       0 |      100 |       0 |       0 | 20-74
  burger-config.ts                       |       0 |      100 |     100 |       0 | 21
  burger-data.service.ts                 |       0 |        0 |       0 |       0 | 8-28
  burger-director.ts                     |       0 |      100 |       0 |       0 | 8-24
```

Figure 2.37: *Coverage result after task run using npm or yarn*

This report contains full information about coverage and e2e results. Coverage - is the information about how many tests exist for each file and function is exist in your system using the percentage system for measuring. *(Figure 2.38)*:

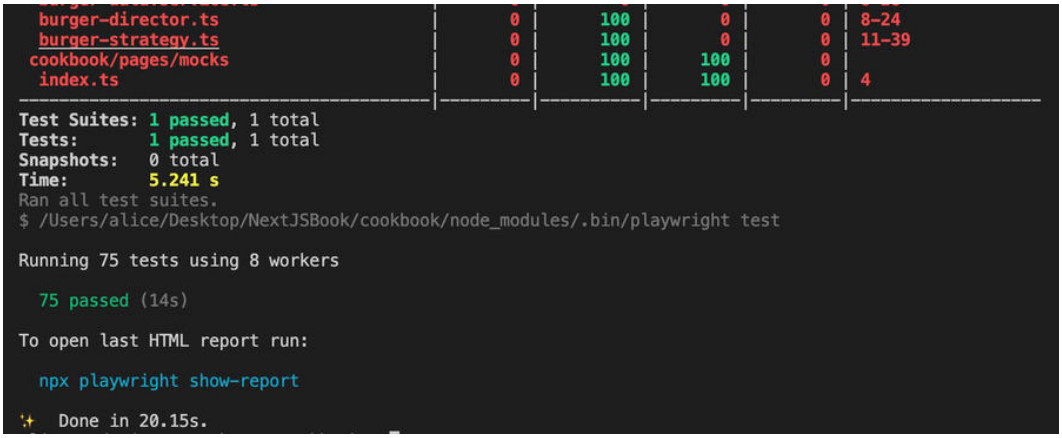


Figure 2.38: *Test report with all passed tests*

Writing your first component in a test-first way

To start developing we will require to create the test file in **"__tests__"** folder. We will agree that next page that we will create will be the list of recipes. So the code in our new file will look like this as shown in *Figure 2.39:*

```
import React from 'react'
import { render, screen } from '@testing-library/react'
import RecipeList from '../pages/recipe-list.page'

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

      const heading = screen.getByRole('heading', {
        name: /Recipe list/i,
      })

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

Figure 2.39: *Test code to check text in the component*

Here we are created the test that expects page to be rendered with heading that contain text **"Recipe list".** Start the test with **"yarn test"** command in your console. The result will look like this:

```
$ jest
 PASS  __tests__/index.test.jsx
 FAIL  __tests__/recipe.test.jsx
  ● Test suite failed to run

    Cannot find module '../pages/recipe-list.page' from '__tests__/recipe.test.jsx'

      1 | import React from 'react'
      2 | import { render, screen } from '@testing-library/react'
    > 3 | import RecipeList from '../pages/recipe-list.page'
        | ^
      4 |
      5 | describe('RecipeList', () => {
      6 |     it('renders a heading', () => {

      at Resolver._throwModNotFoundError (node_modules/jest-resolve/build/resolver.js:491:11)
      at Object.<anonymous> (__tests__/recipe.test.jsx:3:1)

Test Suites: 1 failed, 1 passed, 2 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.989 s, estimated 1 s
```

Figure 2.40: *The failing result of the test that we created*

The test is failing as we do not have such page component yet. So next steps to follow this development way is to create the component that will fit requirements inside. For example, we will create the page component like the following figure:

```
import type { NextPage } from 'next'
import Link from 'next/link'

const RecipeList: NextPage = () => {
  return (
    <div>
      <h1>Hello there ! This is the recipes page of
Cooks</h1>
  )
}

export default RecipeList
```

Figure 2.41: *Component that was expected in the test*

And now if you start the test again the test will fail with another message as shown in *Figure 2.42*:

```
$ jest
 PASS  __tests__/index.test.jsx
 FAIL  __tests__/recipe.test.jsx
  ● RecipeList › renders a heading

    TestingLibraryElementError: Unable to find an accessible element with the role "heading" and name `/Recipe list/i`

    Here are the accessible roles:

      heading:

      Name "Hello there ! This is the recipes page of CookBook":
      <h1 />

      --------------------------------------------------

    Ignored nodes: comments, <script />, <style />
    <body>
      <div>
        <div>
          <h1>
            Hello there ! This is the recipes page of CookBook
```

***Figure 2.42**: The failing result of the test*

So, we are having an error message about the heading that not fits the expectations that we are created. This explained with the following code line:

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

***Figure 2.43**: Test code line that triggers the error*

In simple words the testing process is contains creating the logic of what expected as a result of component or function job. Here we see that we expect this element to be rendered in the component as shown in *Figure 2.44:*

```
const heading = screen.getByRole('heading', {
        name: /Recipe list/i,
 })
```

***Figure 2.44:** Test code with text that should be on the page*

So as a result of it we are getting error as its not exists. Let us follow the test requirements and add the required text into element like this:

```tsx
import type { NextPage } from 'next'
import Link from 'next/link'

const RecipeList: NextPage = () => {
  return (
    <div>
      <h1>Recipe list</h1>
    </div>
  )
}

export default RecipeList
```

Figure 2.45: *Updated component code*

Now the result of the test will look like the following figure:

```
$ jest
 PASS  __tests__/recipe.test.jsx
 PASS  __tests__/index.test.jsx

Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.02 s
Ran all test suites.
✨  Done in 2.09s.
```

Figure 2.46: *Success result of the test code run*

All tests are passed and having a green color. In the end we are having the correct development flow where we create the requirements for the code before we create the code itself.

Next step is to cover after rendering the process and create the E2E test. To do it create the file **"`recipe-list.spec.ts`"** in "`e2e`" folder that will contain the following code:

```tsx
import { test, expect, Page } from '@playwright/test';

test.describe('Recipe page result', () => {
    test('should open the created page in browser', async ({ page }) => {
        await page.goto('http://localhost:3005/recipe-list')
        await expect(page.locator('h1')).toHaveText('Recipe list will fail')
    })
})
```

Figure 2.47: *E2E test example with an open page action*

Now we can check the test by typing "`yarn test:e2e`" in the console. The result will look as shown in *Figure 2.48:*

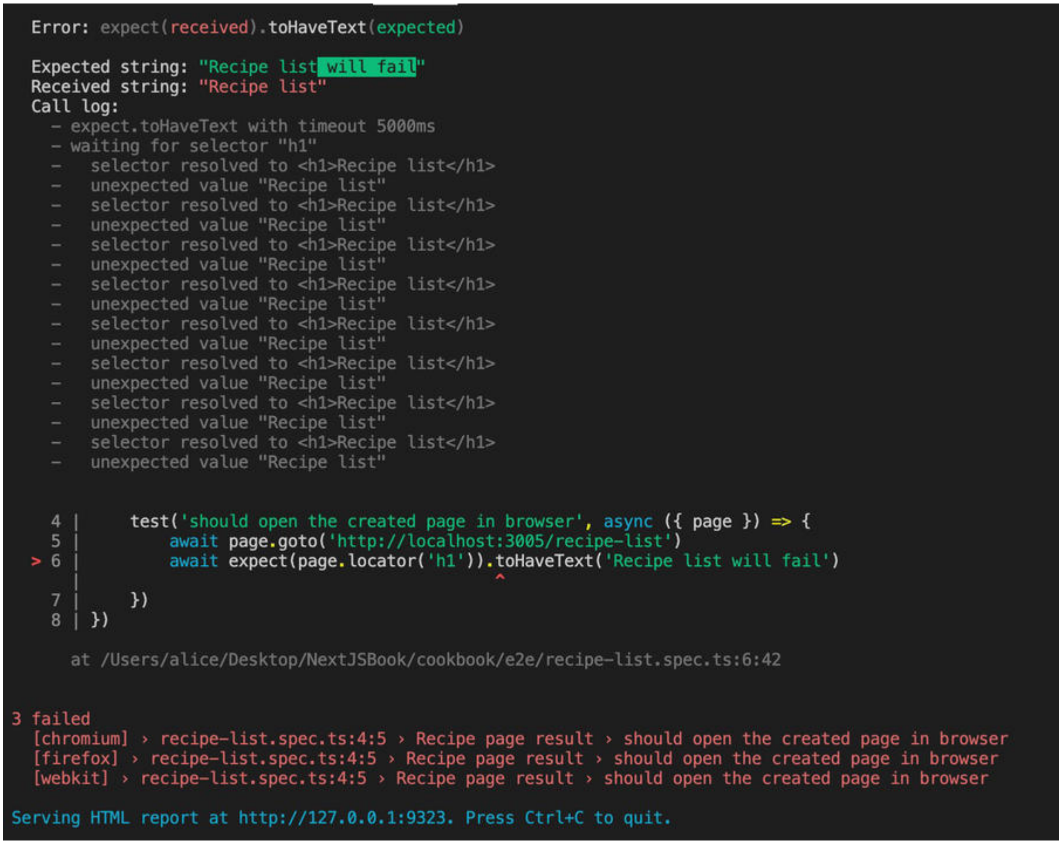


Figure 2.48: *Failing test result for the E2E test*

The browser will be opened and the result page will appear as shown in *Figure 2.49:*

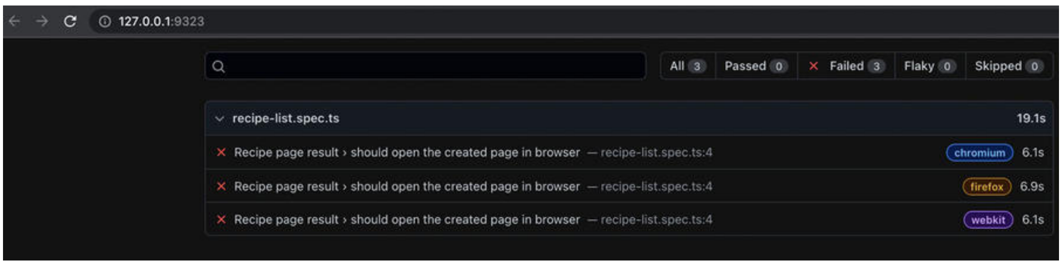


Figure 2.49: *E2E test reporting page*

As we can see the test is failed. But we made it ourselves just to show how the process will look like this. Let us fix this by typing the correct text in the expected area as depicted in *Figure 2.50:*

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

test.describe('Recipe page result', () => {
    test('should open the created page in browser', async ({ page }) => {
        await page.goto('http://localhost:3005/recipe-list')
        await expect(page.locator('h1')).toHaveText('Recipe list')
    })
})
```

***Figure 2.50**: Updated text that expects correct data*

Now the result will be like *Figure 2.51:*

```
yarn run v1.22.18
$ yarn playwright test
$ /Users/alice/Desktop/NextJSBook/cookbook/node_

Running 3 tests using 3 workers

  3 passed (2s)

To open last HTML report run:

  npx playwright show-report

✨  Done in 2.77s.
```

***Figure 2.51:** Success result of the test*

Hence, all the tests are green now.

Conclusion

There is no special purpose to not to think about the code quality in the very beginning. In most of the cases, software developers think that if we use Agile then we can just do a lot of refactoring all the time. It is true on one side, but on the other side, hardly structured code base is not open to refactoring and scalability.

We can hardly recommend using the knowledge from this chapter to use in your next new application. It will give you a hundred steps forward to create a very clean and scalable codebase in the future.

In the next chapter we are going to learn what is authorization from the application perspective and how to design and start implementing a login form in our application.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 3
Authorization in a glance with NextJS

Introduction

Any modern application that is possible to invent today requires a personal user area. To achieve this - we will need authorization. There are several possible ways to create and implement auth system into your application. There are also super-simple solutions that require only implementing the auth system into the application (like AWS Amplify or Google Firebase) as well as the solutions that involve coding and architecture.

To understand how exactly the authorization is working, we will do everything manually from the raw stage with idea creation to the final realization.

Structure

- Creating the authorization form
  - How to mock your first component using pencil and your ideas?
  - How to split component into generic components?
  - Separating global styles form local styles for any component
  - Creating the code logic structure for authorization form
  - How to write tests for authorization form?

- From unit test to NextJS component
  - How to follow the TDD way in creating the components?
  - How to debug tests while developing?
- Advantages of REST way authorization
- Advantages of GraphQL authorization

Objectives

In this chapter we will start working with authorization and touch the topics what is the authorization itself and how to start creating the authorization from the start to ready application. Also we will be introduced how to use it with the test driven development and compare the http connection ways using REST and GraphQL ways of connections

Creating the authorization form

From the user perspective, there are not many ways to get into the personal area not using the special pages or interfaces for it. To pass the user into the personal area we will need the authorization form in our system that will contain the login and password fields log in.

There is no reason to start coding immediately when you get the task to create something. Before that, we will need to do some preparations that will help us and save a lot of time. Always remember: the code is only the realization, what matters is your logic and idea.

Mocking your first component using a pencil and your ideas

Let us start creating our first component and for the first step, we do not need anything except something to draw and a quiet place to think. You can use any comfortable tool as nowadays there are a lot of free tools out there for example . Figma, but at the current state, we can just use the pencil and paper.

Let us create some requirements that will help us to proceed with everything. We will call it **General requirements:**

- The form will use a separate page
- All form states will be provided on one page
  - Possible form states

        - Login form
        - Login form with error
        - Welcome layout with redirection and count down in 5 seconds
- The form will contain 2 input fields
    - Login field
        - Should take only English letters
        - 3 letters minimum
        - 100 letters maximum
        - All HTML special characters must be removed from the input
        - The error about validation should be shown if we change focus or click on a submit button (that also changing of focus)
        - The input field should have a "text" type
    - Password field
        - Should take only English letters
        - 8 letters minimum
        - 50 letters maximum
        - All HTML special chars must be removed from the input
        - Data inside should contain one Capital letter and one number
        - The error about validation should be shown if we change focus or click on a submit button (that also changing of focus)
        - The input field should have a "password" type
    - Submit button
        - The button is always active
        - If we do request into API we should disable the button and show animation for the call
        - Click on the button should contain prevent default logic to prevent bubbling clicks outside of the form
        - The background color should be #84DCC6
- The form should be vertically and horizontally aligned to the page center
- Form max-width should be 800px
- Form background color should be #EEF7FB
- Form error color should be #FF0A0A

Now when we have all requirements, we can draw some mockups for our authorization system. In the *Figure 3.1* we see the mock up for the login form that will be used to enter the private user area.

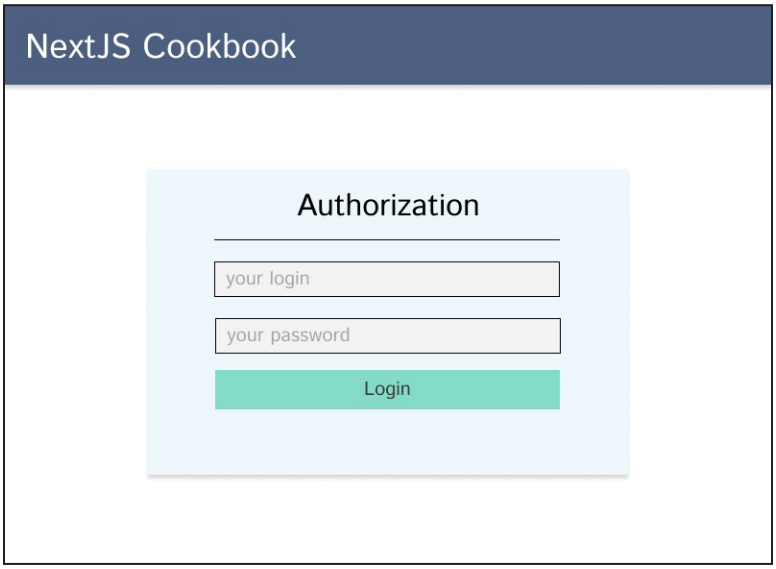


Figure 3.1: *Login form mock*

When we did some mockups for the login page we can proceed with requirements and create a layout for the errors that will look like this:

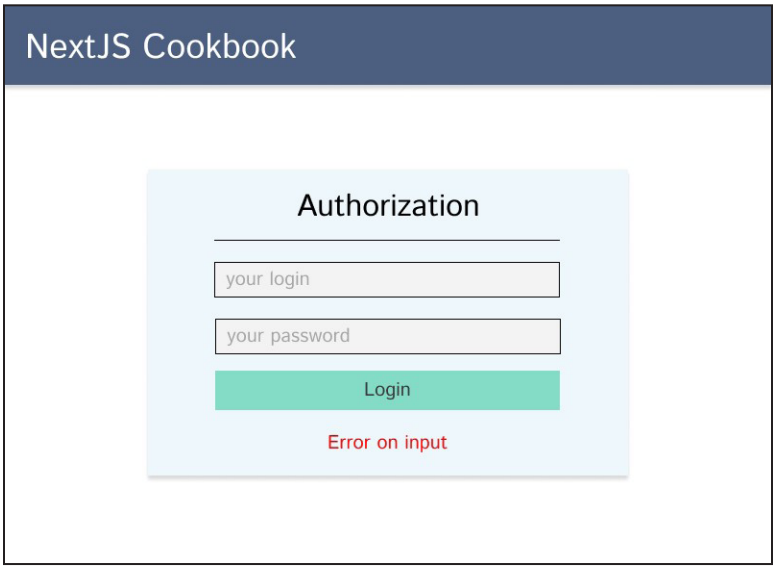


Figure 3.2: *Login form mock in case if there are some errors in the credentials*

The last one if everything is correct will look like this:

Figure 3.3: *Mock for success authorization screen*

Now we have everything to start coding. But before that, we will look at the Atomic design system to create a better structure for the UI of the application.

Splitting components into generic components

What is atomic in general? Atomic Design is a methodology created by Brad Frost seeking to provide direction on building interface design systems more deliberately and with explicit order and hierarchy.

There are several levels of elements that is presenting the design level for each component that contain level names: **`Atoms`**, **`Molecules`**, **`Organisms`**, **`Templates`**, **`Pages`** all appear like in *Figure 3.4:*

Figure 3.4: *Atomic design methodology scheme*

Let`s describe what *Figure 3.4* means:

- Atoms are the building blocks of al matter, and in the context of Atomic Design, atoms are the smallest, most basic elements of a user interface. Examples of atoms include individual HTML elements like buttons, form inputs, and icons.
- Molecules are collections of atoms that have been grouped together to create more complex UI elements. Examples of molecules include forms, search bars, and cards.
- Organisms are combinations of molecules that form distinct sections of an interface, such as headers, footers, and navigation menus.
- Templates are higher-level representations of how an interface might be structured, and they typically include a combination of organisms, molecules, and atoms. Examples of templates include homepage templates and product detail page templates.
- Pages are the final level of Atomic Design, and they represent the specific instances of templates that are used to deliver content to the end user.

As NextJS is using this pattern for the pages we will use it for the smaller components to follow the good practice of application creation with React (and NextJS in general).

So following this system we will have this structure of possible components. We will call it **UI requirements**:

- Atoms
  - Login input
    - Should have a placeholder: Enter your login
  - Password input
    - Should have a placeholder: Enter your password
  - Submit button
    - Should have a label: Login
- Molecules
  - Login form and Welcome message layout
    - Should contain form
    - Should contain error field
    - On success, the form should be switched to welcome text
- Organisms
  - Authorization layout that centered on page CSS rules

- Templates
  - o System messages and auth
- Pages
  - o Login page

As we do not have so many different elements at the current state we will not use Organisms and Templates for the current example as it will be just a wrapper for the component that will not do anything. But we will extend it in the next chapters. For now, the structure of our application should appear like *Figure 3.5:*

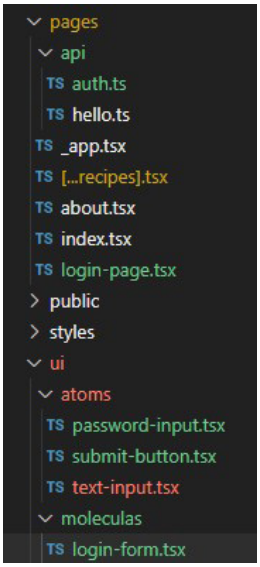


Figure 3.5: *Structure of the project that will represent Atomic system*

Separating global styles from local styles for any component

As we are following the Atomic design it is also a good practice to split your styles into a meaningful part to have more convenient maintainability.

To achieve it let us split all styles into four pieces:

- Design tokens
- Global styles
- Utility classes
- Component styles

Design tokens, global styles and utility classes are mostly universal styles for the project not separated for each component. In the **Design tokens,** we will store all possible variables for the application as sizes, colors, margins, and other properties. The file will be named `**variables.scss**`. We already have a file named `**colors.scss**` so all content from it will be moved into variables. For the **Global styles,** we will use the file named `**globals.scss**` that already exists in the system. In global styles we will store typography styles, layouts settings and styles. In the **Utility classes** we will store all possible mixins and functions that can be reused in some components but are not required for all of them so should not be stored in the **globals.scss** file. Let us name the file as `**utilities.scss**`.

And the last one for the **Component styles** we will use the system that we already use as **`<module-name>.module.scss`.** In this style file we will store components specific styles for example, buttons, forms, inputs

Creating the code logic for the authorization form

As we created the requirements for UI before we need also to extend the requirements for the business logic as authorization is not only the login form.

In simple words the authorization should work like as shown in *Figure 3.6:*

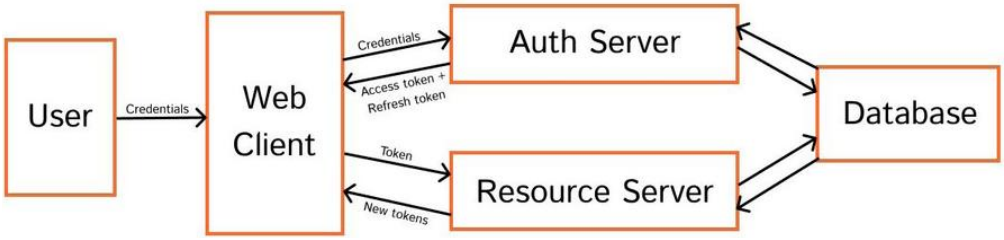


Figure 3.6: *Authorization scheme as high-level architecture*

To access any resource at the resource server we need to have a token. There are two kinds of tokens: **Access token** and **Refresh token**. The access token is responsible, literally for the access to the resource and the refresh tokens are required in case when the access token is expired. What is meant here is to have the possibility to access any web page or data we need to call it with a special key (Access token). But for the security reason this token should be expired in some amount of time (for example. each 1 hour the token becomes useless). After that API should regenerate the token using also some permission to do it (Refresh token). And now we can use the system as long as we want.

Token should not be hashed, but rather encrypted, as hashing is a one-way function that cannot be reversed. Encryption, on the other hand, can be decrypted with a key, allowing the token to be verified and decoded by the resource server.

Let us extend our requirements and add some points about API responses. We will call it **Business logic requirements**:

- API should answer with the hashed string that we will call token if login and password are correct. Answer code – 200.
- If the login or password is incorrect answer code should be 401 (which means literally that it is unauthorized). This code is standard for unauthorized request.
- The refresh token should be used to regenerate a new token every 60 minutes.
- If none of the tokens are valuable - redirect the user to the login page.

This is a good fallback option to ensure that users cannot continue to access resources without valid credentials. However, it is important to consider how you will handle situations where the user's session has timed out or the refresh token has expired. In these cases, we can provide a specific error message or prompt the user to log in again. In our case we will not do that as its out of the scope of this book

Using all requirements we can start with coding. But before that also using requirements we need to cover our code with all possible tests.

Writing tests for the authorization form

The application requirements are the best approach to start with the test development. But first of all, let us talk a bit about what is tests means.

In software development, there are not many ways to achieve the point when we can understand that an application or even a small part of it is ready and up and running. On one hand, we can check the requirements manually each time when we do changes or updates. On other hand, we can automate this process and not waste time on paperwork. To save time and have more safety we should create the tests that cover our requirements point by point.

Let us look at our **General requirements** from the upper scope. The requirements it itself is produced in the same pattern as we need to create the test:

```
<Name of the logical part> => <should have current result>
```

For example, we have the requirement **"Possible state of the form => Login form with error".** That means that we need to create the test that will cover provided logic like this:

```
If we have the error in the form => This error should be stored in the form state
```

Pretty simple and this is a good part of testing the application before coding. It is pretty simple as it is covering the provided requirements from the human text into the programming language.

From unit test to NextJS component

In the development flow, there are no strict rules on what requirements should be made first. We will stick to the rule "From smaller to bigger" and the order of test development will look like this:

- UI requirements
- General requirements
- Business logic requirements

The motivation for this ordering is that we will produce tests from smaller components to bigger logical parts.

Following the TDD way in creating components

We will start with UI requirements and will follow them in the test driven development way of creating. **Atoms** say that we need 3 elements in there so let us make the test that will expect these elements exist. Also, we will create one more helper file that will contain the placeholders for all possible inputs. These placeholders we will use as a query selector for the test (you could use any identification that is more convenient for you, this is not mandatory).

In the **core** folder, we will create the **configs** folder with the index file inside. Put this code inside to create the configuration for the placeholders and the labels as shown in *Figure 3.7:*

```
enum Placeholders {
    TEXT_INPUT = 'Enter your login',
    PASSWORD_INPUT = 'Enter your password',
}

enum Labels {
    SUBMIT = 'Login'
}

export { Placeholders, Labels }
```

Figure 3.7: *Application configuration for the authorization*

These enums already follow the UI requirements. Next, we need to create the unit test file in the **`__tests__` `folder`**. In this file, we will need to have this code to start:

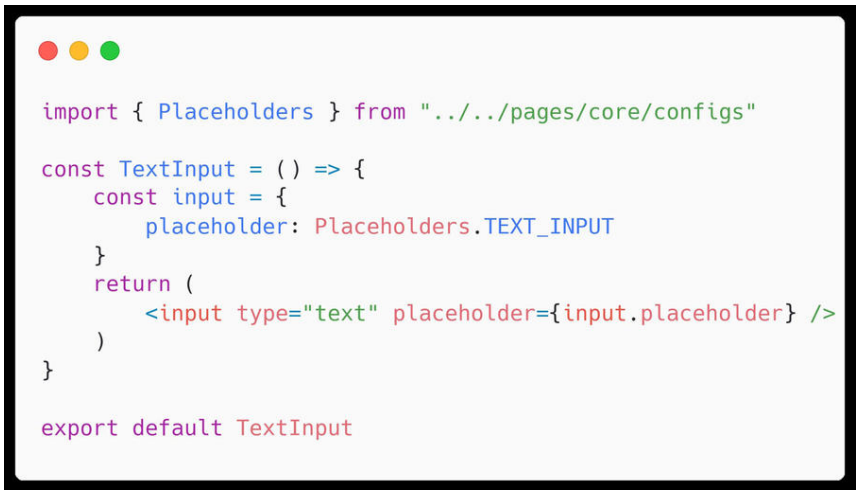
```
import React from 'react'
import { render, screen } from '@testing-library/react'
import TextInput from '../ui/atoms/text-input'
import PasswordInput from '../ui/atoms/text-input'
import SubmitButton from '../ui/atoms/submit-button'
import { Labels, Placeholders } from '../pages/core/configs'

describe('UI inputs must render properly', () => {
    it('renders a text input', () => {
      render(<text-input />)
      const input = screen.getByPlaceholderText(Placeholders.TEXT_INPUT)
      expect(input).toBeInTheDocument()
    })

    it('renders a password input', () => {
      render(<password-input />)
      const input = screen.getByPlaceholderText(Placeholders.PASSWORD_INPUT)
      expect(input).toBeInTheDocument()
    })

    it('renders a submit button', () => {
      render(<submit-button />)
      const submit = screen.getByText(Labels.SUBMIT)
      expect(input).toBeInTheDocument()
    })
  })
```

Figure 3.8: *Tests for the login form that we will use to create the form*

If we start our tests now all of them will be failed. We expect components that still do not exist in the system. Moreover, even if they will exist they should follow the requirements and contain text inside. That means that we can proceed and follow our first TDD requirements.

In the end our components will look like this:

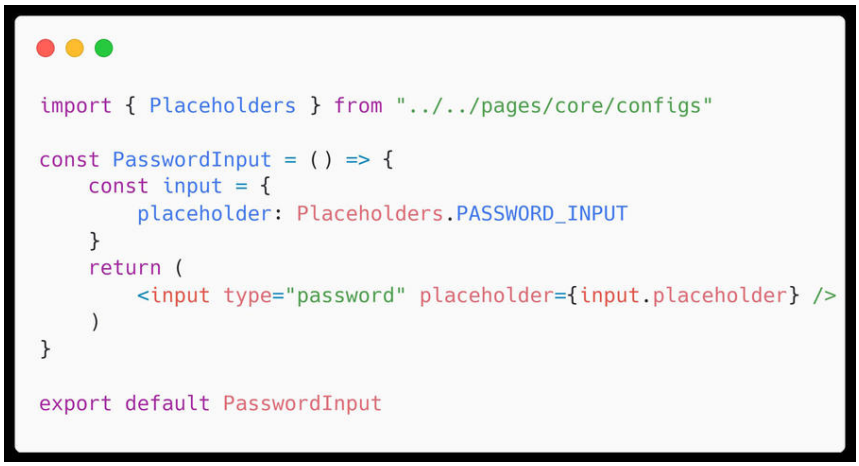
1. Regular input component will contain the code that is presented on the *Figure 3.9:*



```
import { Placeholders } from "../../pages/core/configs"

const TextInput = () => {
    const input = {
        placeholder: Placeholders.TEXT_INPUT
    }
    return (
        <input type="text" placeholder={input.placeholder} />
    )
}

export default TextInput
```

Figure 3.9: *Regular text input component*

2. Password input component will contain the code that is presented on the *Figure 3.10*



```
import { Placeholders } from "../../pages/core/configs"

const PasswordInput = () => {
    const input = {
        placeholder: Placeholders.PASSWORD_INPUT
    }
    return (
        <input type="password" placeholder={input.placeholder} />
    )
}

export default PasswordInput
```

Figure 3.10: *Password input component*

3. Submit button component will contain the code that is presented on the *Figure 3.11*

```
import { Labels } from "../../pages/core/configs"

const SubmitButton = () => {
    const input = {
        label: Labels.SUBMIT
    }
    return (
        <button type="submit">{input.label}</button>
    )
}

export default SubmitButton
```

Figure 3.11: *Submit button component*

Now, if we run the tests all of them will be passed as we have required components with required text inside of them like shown in *Figure 3.12:*

```
$ yarn test
yarn run v1.22.17
$ jest
 PASS  __tests__/index.test.jsx
 PASS  __tests__/ui.test.tsx

Test Suites: 2 passed, 2 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        3.671 s
Ran all test suites.
Done in 5.14s.
```

Figure 3.12: *Console result with all tests passed*

Let us proceed with **Molecules** and create the form component that should contain our atoms inside and render the form. We will create the test that will check if the component renders and contains the required components.

The logical trick is that we need to call the same test cases to check if the element is exist in the layout. To optimize it let us create the test object that will contain elements that will be reused like the following figure:

```
const testObject: {[key: string]: any} = {
  isTextInput: (screen) => screen.getByPlaceholderText(Placeholders.TEXT_INPUT),
  isPasswordInput: (screen) => screen.getByPlaceholderText(Placeholders.PASSWORD_INPUT),
  isSubmitButton: (screen) => screen.getByText(Labels.SUBMIT)
}
```

Figure 3.13: *Test code organization to optimize tests*

We will use functions instead of direct calls as we do not have any rendered elements on the screen at the start of the test, we need to render something first. To avoid any errors here we can use the function that will take a screen as a parameter and check on the call stage. Also we will need the function that will cover any expectation depending on what object element is currently covered. That function will look like the following figure:

```
const expectation =  (element) => expect(element).toBeInTheDocument()
```

Figure 3.14: *Expectation sentence after optimization*

After that, we can change our previous tests to use this object and function as shown in *Figure 3.15:*

```
describe('UI inputs must render properly', () => {
  it('renders a text input', () => {
    render(<TextInput />)
    expectation(testObject.isTextInput(screen))
  })

  it('renders a password input', () => {
    render(<PasswordInput />)
    expectation(testObject.isPasswordInput(screen))
  })

  it('renders a submit button', () => {
    render(<SubmitButton />)
    expectation(testObject.isSubmitButton(screen))
  })
})
```

Figure 3.15: *Resulting test code that can work with any kind of input components*

The full file with the test will also contain the new test that will contain the logic to check if all elements exist in the layout: (*Figure 3.16)*

```
import React from 'react'
import { render, screen } from '@testing-library/react'
// Atoms
import TextInput from '../ui/atoms/TextInput'
import PasswordInput from '../ui/atoms/PasswordInput'
import SubmitButton from '../ui/atoms/SubmitButton'
// Molecules
import LoginForm from '../ui/molecules/LoginForm'
import { Labels, Placeholders } from '../pages/core/configs'

const testObject: {[key: string]: any} = {
  isTextInput: (screen) => screen.getByPlaceholderText(Placeholders.TEXT_INPUT),
  isPasswordInput: (screen) => screen.getByPlaceholderText(Placeholders.PASSWORD_INPUT),
  isSubmitButton: (screen) => screen.getByText(Labels.SUBMIT),

}

const expectation =  (element) => expect(element).toBeInTheDocument()

describe('UI inputs must render properly', () => {
  it('renders a text input', () => {
    render(<TextInput />)
    expectation(testObject.isTextInput(screen))
  })

  it('renders a password input', () => {
    render(<PasswordInput />)
    expectation(testObject.isPasswordInput(screen))
  })

  it('renders a submit button', () => {
    render(<SubmitButton />)
    expectation(testObject.isSubmitButton(screen))
  })
})

describe('Form should be rendered properly', () => {
  it('renders login form', () => {
    render(<LoginForm />)
    const testKeys = Object.keys(testObject)
    if(Array.isArray(testKeys) && testKeys.length > 0) {
      testKeys.forEach((test: string) => {
        expectation(testObject[test](screen))
      })
    }
  })
})
```

Figure 3.16: *Full code of the test to cover all possible inputs for the login form*

As we started to test it - it will be failed as there are no such elements in the layout. So we must create them in the **`LoginForm`** file to follow the test requirements like *Figure 3.17:*

```
import TextInput from '../atoms/TextInput'
import PasswordInput from '../atoms/PasswordInput'
import SubmitButton from '../atoms/SubmitButton'

const LoginForm = () => {
    return (
        <section>
            <div>
                <TextInput />
            </div>
            <div>
                <PasswordInput />
            </div>
            <div>
                <SubmitButton />
            </div>
        </section>
    )
}

export default LoginForm
```

Figure 3.17: *Login form starter to pass the tests*

Let us update the test object as we have a requirement regarding the error layout existing in the form.(*Figure 3.18):*

```
const testObject: {[key: string]: any} = {
  isTextInput: (screen) => screen.getByPlaceholderText(Placeholders.TEXT_INPUT),
  isPasswordInput: (screen) => screen.getByPlaceholderText(Placeholders.PASSWORD_INPUT),
  isSubmitButton: (screen) => screen.getByText(Labels.SUBMIT),
  isErrorField: (screen) => screen.getByTestId(TestIDs.ERROR)
}
```

Figure 3.18: *Adding the error element into the testing objects*

Here we will use a special identification that will be used only for tests. In the form this layout will look like the following figure:

```
import TextInput from '../atoms/TextInput'
import PasswordInput from '../atoms/PasswordInput'
import SubmitButton from '../atoms/SubmitButton'
import { TestIDs } from '../../pages/core/configs'

const LoginForm = () => {
    const errorTestID = TestIDs.ERROR
    return (
        <section>
            <div>
                <TextInput />
            </div>
            <div>
                <PasswordInput />
            </div>
            <div>
                <SubmitButton />
            </div>
            <div data-testid={errorTestID}></div>
        </section>
    )
}

export default LoginForm
```

Figure 3.19: *Adding the error element into the form*

Also do not forget to update the configs with a new enum that will contain the ids for the tests. Put **`ERROR = 'error'`** inside of it.

Now we can try to test it again and the tests should also pass all the requirements: (*Figure 3.20*)

```
Test Suites: 2 passed, 2 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        5.023 s
Ran all test suites.
Done in 7.08s.
```

Figure 3.20*: Success result for the application tests*

For now, we cannot cover the style test without an actual render of the component but we will come back to it in the next chapters where we will cover the E2E test topic for the application. Also, we will come back to extend these tests in the chapter about state management.

We can proceed with **General requirements** and extend our tests to fit them too. The only possible to test without render cases is a type of language that will be in the input value and the maximum length of the value. To complete our tests with it we need to extend our test file and add some logic to it. First, let us add the function that will generate a random string with characters and numbers like *Figure 3.21:*

```
const makeLogin = (length: number) => {
  var result           = '';
  var characters       = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
  var charactersLength = characters.length;
  for ( var i = 0; i < length; i++ ) {
    result += characters.charAt(Math.floor(Math.random() *
charactersLength));
 }
 return result;
}
```

Figure 3.21: *Function to generate a random string*

Now we can put this value into our input to check what is inside of it. In the test itself, we need to check - if the language is wrong then the value should not be changed like the following figure:

```
fireEvent.change(input, { target: { value: 'тест' } })
expect(input).toHaveDisplayValue('')
```

Figure 3.22: *This is how we will check the type of language. In this example, we use the Russian word. Only English words are acceptable for the login*

Now as you see the test will check if the language is not English then it should not change the value of the input. For now, the test will fail as we do not have such logic in the input. We need to extend our text-input component to provide such logic as this in *Figure 3.23:*

```
if(isLetter(value) && value.length <= maxValueLength) {
   setValue(value)
}
```

Figure 3.23: *Expression to follow test requirements*

Having this logic in the component we will check if the requirements fit then we can change the input value. As you can see, we did not provide the function **`isLetter`** and we need to provide it. Let us create the folder in our core that will be named **`"utils.ts".`** This file will contain all utility functions and helpers for our application. This file will be like this as *Figure 3.24:*

```
const isLetter = (str: string) => {
    if (str.length === 0) return true
    return /[A-Za-z0-9 ]/.test(str)
}

export { isLetter }
```

Figure 3.24: *Test to cover that string is not contain any special characters*

As you can see we will check if the provided string does not contain any characters that are not English or numbers or whitespace. The updated input component will look like this if we will implement this logic inside: (*Figure 3.25):*

```
import { ChangeEvent, useState } from "react";
import { Placeholders } from "../../pages/core/configs"
import { isLetter } from "../../pages/core/utils";

const TextInput = ({ onLoginEnter, id } : any) => {
    const input = {
        placeholder: Placeholders.TEXT_INPUT
    }
    const maxValueLength = 100

    const [value, setValue] = useState('')

    const onChangeHandler = (event: ChangeEvent<HTMLInputElement>) => {
        const value = event.target.value
        if(isLetter(value) && value.length <= maxValueLength) {
            setValue(value)
        }
        onLoginEnter(value)
    }

    return (
        <input id={id} type="text" onChange={onChangeHandler} placeholder={input.placeholder} value={value} />
    )
}

export default TextInput
```

Figure 3.25: *Full text of the component for the login*

And the updated tests will look like *Figure 3.26:*

```
it('Should render login input and check that it can take only Englsh letters', () => {
    render(<TextInput onLoginEnter={(value) => value}/>)

    const input = testObject.isTextInput(screen)

    expectation(testObject.isTextInput(screen))

    fireEvent.change(input, { target: { value: 'おはようございます' } })
    expect(input).toHaveDisplayValue('')

    const generatedLogin = makeLogin(101)

    fireEvent.change(input, { target: { value: generatedLogin } })
    expect(input).toHaveDisplayValue('')
  })
```

Figure 3.26: *Full text for the test of login input component*

The login input is fully covered with possible unit tests and can be safely used in our application. Now we need to use the same way to update the password input. The component will look like the following figure:

```
import { useState } from "react"
import { Placeholders } from "../../pages/core/configs"
import { isLetter } from "../../pages/core/utils"

const PasswordInput = () => {
    const input = {
        placeholder: Placeholders.PASSWORD_INPUT
    }
    const maxValueLength = 50
    const [value, setValue] = useState('')

    const onChangeHandler = (event: Partial<any>) => {
        const value = event.target.value
        if(isLetter(value) && value.length <= maxValueLength) {
            setValue(value)
        }
    }

    return (
        <input type="password" onChange={onChangeHandler} placeholder={input.placeholder} value={value} />
    )
}

export default PasswordInput
```

Figure 3.27: *Full text of password component*

And finally, the UI tests file will look like *Figure 3.28:*

```
import React from 'react'
import { render, screen, fireEvent } from '@testing-library/react'
// Atoms
import TextInput from '../ui/atoms/TextInput'
import PasswordInput from '../ui/atoms/PasswordInput'
import SubmitButton from '../ui/atoms/SubmitButton'
// Molecules
import LoginForm from '../ui/molecules/LoginForm'
import { Labels, Placeholders, TestIDs } from '../pages/core/configs'

const testObject: {[key: string]: any} = {
  isTextInput: (screen) => screen.getByPlaceholderText(Placeholders.TEXT_INPUT),
  isPasswordInput: (screen) => screen.getByPlaceholderText(Placeholders.PASSWORD_INPUT),
  isSubmitButton: (screen) => screen.getByText(Labels.SUBMIT),
  isErrorField: (screen) => screen.getByTestId(TestIDs.ERROR)
}

const makeLogin = (length: number) => {
  var result           = '';
  var characters       = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
  var charactersLength = characters.length;
  for ( var i = 0; i < length; i++ ) {
    result += characters.charAt(Math.floor(Math.random() *
charactersLength));
 }
 return result;
}

const expectation =  (element) => expect(element).toBeInTheDocument()

describe('UI inputs must render properly', () => {
  it('renders a text input', () => {
    render(<TextInput />)

    const input = testObject.isTextInput(screen)

    expectation(testObject.isTextInput(screen))

    fireEvent.change(input, { target: { value: 'тест' } })
    expect(input).toHaveDisplayValue('')

    const generatedLogin = makeLogin(101)

    fireEvent.change(input, { target: { value: generatedLogin } })
    expect(input).toHaveDisplayValue('')
  })

  it('renders a password input', () => {
    render(<PasswordInput />)
    const input = testObject.isPasswordInput(screen)

    expectation(input)

    fireEvent.change(input, { target: { value: 'тест' } })
    expect(input).toHaveDisplayValue('')

    const generatedLogin = makeLogin(51)

    fireEvent.change(input, { target: { value: generatedLogin } })
    expect(input).toHaveDisplayValue('')
  })

  it('renders a submit button', () => {
    render(<SubmitButton />)
    expectation(testObject.isSubmitButton(screen))
  })
})

describe('Form should be rendered properly', () => {
  it('renders login form', () => {
    render(<LoginForm />)
    const testKeys = Object.keys(testObject)
    if(Array.isArray(testKeys) && testKeys.length > 0) {
      testKeys.forEach((test: string) => {
        expectation(testObject[test](screen))
      })
    }
  })
})
```

Figure 3.28: *Full text of the test file after all updates*

And if we try to start them all tests will be passed as illustrated in *Figure 3.29:*

```
$ jest
 PASS  __tests__/index.test.jsx
 PASS  __tests__/ui.test.tsx

Test Suites: 2 passed, 2 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        3.649 s
Ran all test suites.
Done in 4.79s.
```

Figure 3.29: *Success result for the tests*

Debugging tests while developing

Sometimes we need to do some debugging as we do in any JS code. The difference is that doing the test with Jest we cannot see anything in the browser and do not have a console. But this problem is easy to solve with the VS Code extension that can be grabbed here by this link **https://marketplace.visualstudio.com/items?itemName=Orta.vscode-jest**. Wising this extension we can put the breakpoints in the test like *Figure 3.30:*

```
30
31  describe('UI inputs must render properly', () => {
32    it('renders a text input', () => {
33      render(<TextInput />)
34
35      const input = testObject.isTextInput(screen)
36
37      expectation(testObject.isTextInput(screen))
38
39      fireEvent.change(input, { target: { value: 'тест' } })
40      expect(input).toHaveDisplayValue('')
41
42      const generatedLogin = makeLogin(101)
43
44      fireEvent.change(input, { target: { value: generatedLogin } })
45      expect(input).toHaveDisplayValue('')
46    })
```

Figure 3.30: *Debugging points that will trigger stop at point*

And after that, we will have the possibility to do a debug by selecting it in the dropdown menu like the following figure:

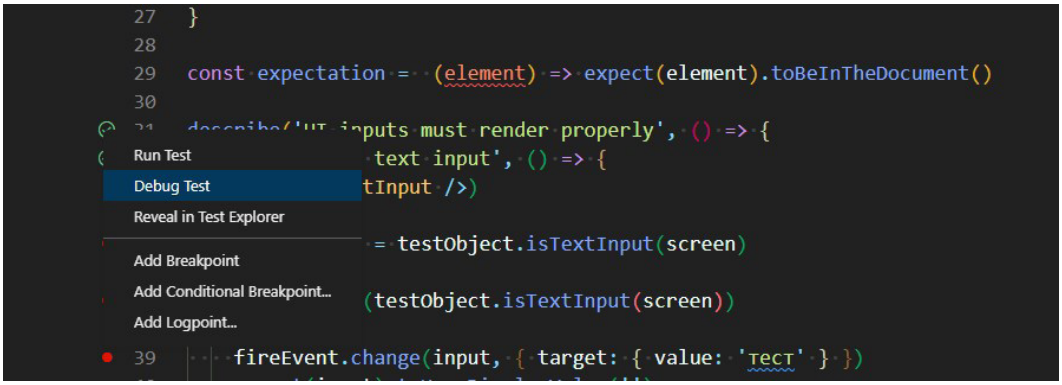


Figure 3.31: *This is how you can enter debug mode*

If you will start it then the plugin will generate the interface with debugging tools to follow all possible debugging steps. You can see it at the top and right part of the screen in the following image:

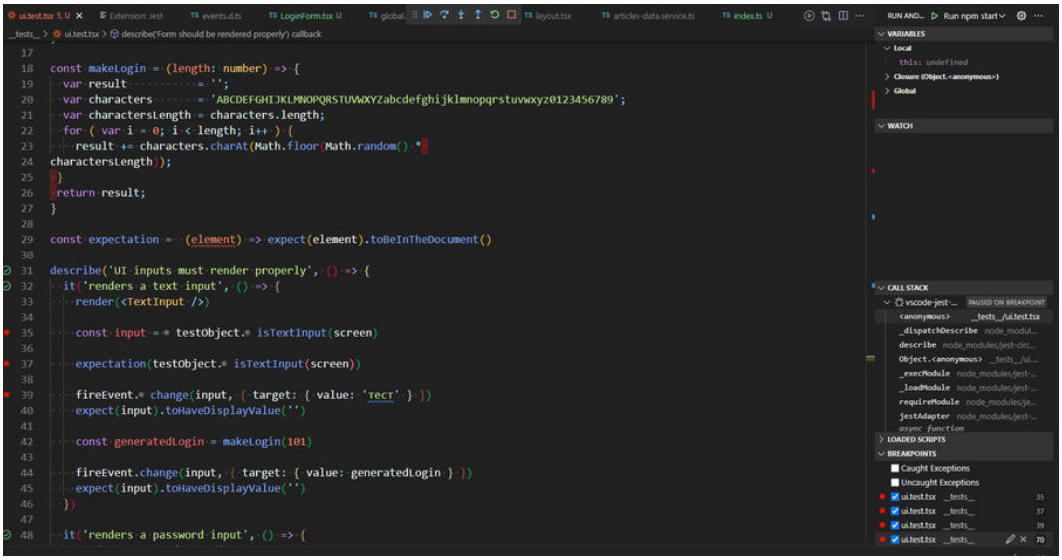


Figure 3.32: *This is how debug break points are looks like*

The stop on a breakpoint will provide you with all possible information to figure out how to proceed with the current breakpoint like *Figure 3.33:*

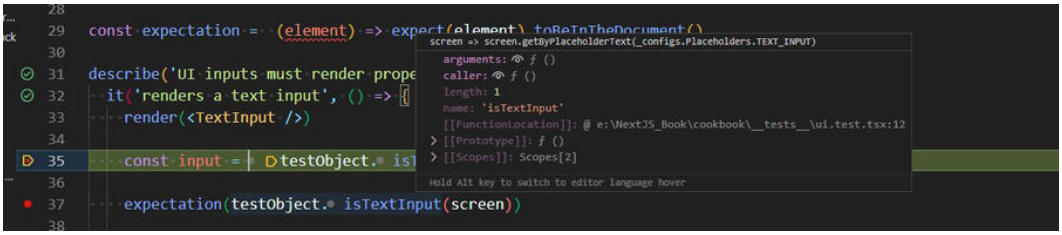


Figure 3.33: *Information that is provided at the break points*

Choosing the next steps way

The authorization itself is not only the form to login but the whole system environment that is connecting a frontend with API and even in the frontend, there are a lot of things that we need to cover. In the next chapter, we will proceed with it but now let us discuss modern ways of API construction as in NextJS we can create full-stack applications using any possible way of the realization.

Advantages of the REST way authorization

Here is a pretty simple answer for the advantages:

- It is easy to monitor as there is not only 200 OK answer from the API
- It is possible to create a micro-service architecture and scale your API service
- It is possible to cache your requests
- Do not require additional software and can be easily implemented using NextJS server-side possibilities (as it is regular NodeJS application) and because of that can be done even without a database in the system.

On the other hand, for the client-based applications, we could do a lot of API changes before going live, and all these changes will require changes in the API.

Advantages of the GraphQL way authorization

In the GraphQL way, some points could be critical when we do a decision:

- It automatically syncs all documentation on any schema change so we do not need to document it manually.
- The data can be fetched with one API call instead of multiple calls.

- All schemas can be changed on the fly and do not require complex deployment from both sides.

But it can be overkill for the small apps and also require more servers at the API side as we need an Apollo server. Also, it is easy to DDoS if we miss access somewhere and the hacking software will be able to create the nested call. This point could create a security issue if we miss some points in the deployment stage.

In this book, we will create a universal solution that will use a model pattern that could be used for any kind of API no matter what we choose.

Conclusion

In this chapter, we started with a very important topic that we will use in the whole application that we will develop in this book. Also, the important thing is not only to write a code but to understand how to start and what to do before coding. What to do before creating an application and what safe tests should and must be done before any code is created.

We now know that authorization is not only the login (or log in and registration) forms, it is more than that and requires accuracy and patience to complete the task.

In the next chapter, we will talk about the server side of NextJS and we will create the API for our application.

After that, we will touch on the topic of state management and in this part, we will be able to complete the task with authorization to the end.

As you see, the authorization at the glance will require a lot of knowledge, but no worries - everything will be covered soon. See you in the next parts.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 4
Server-side power of NextJS

With great power comes great responsibility

— Ben Parker

Introduction

In this chapter, we will introduce the part of **NextJS** that makes it dominate in comparison with all possible competitors in front-end development. In this chapter, we will create the backend API that could be used as backend-for-frontend or as the only backend for future applications.

Why would we need this possibility at all? We must always keep in mind that the browser is not hiding the network history so we can see what request was triggered and what host was used for it. That could be a problem with DDoS attacks and XSS attacks and in the end, could produce a data leak that also can become a problem.

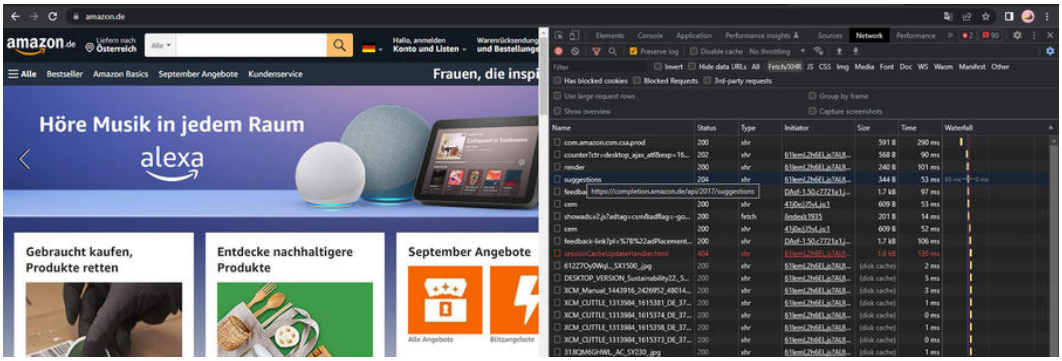


Figure 4.1: *Network activity form the amazon web site*

For example, in *Figure 4.1* we can see all requests on Amazon.de. From now on we can do anything we want with these endpoints. To reduce vulnerability, we could proxy all requests from different places. To achieve that we can use the backend-for-frontend way and not show what requests real we will do and what real response we will get.

Structure

- Using NextJS as an API server
  - o Creating simple NextJS API routing structure
  - o Writing simple API in NextJS
  - o Generating authorization token for user
- Using Singleton, Builder and Strategy patterns in API construction
  - o Baking Singleton for API
  - o Baking Builder for API
  - o Backing Strategy for API
- Using Apollo client for NextJS
  - o Creating models for your NextJS application
- Writing connecting system for Apollo
  - o Reusing API from previous recipe for Apollo?
  - o Setting up Apollo client for NextJS

Objectives

In this chapter we will learn how to use a server side possibilities of the NextJS to use at as a full-stack platform that have a frontend and the backend. We will also reuse knowledge of the design patterns to create the maintainable and scalable code for the application and also will create and connect Apollo server to our application.

Using NextJS as an API server

We do not need any particular setup or configuration to use this feature in the framework. But as you remember we made a configuration for the whole project and have page extension requirements. We can use this extension for API fails but to make them more readable let us add something into the configuration in the **`next.config.js`** file as shown in *Figure 4.2:*

```
const nextConfig = {
  reactStrictMode: true,
  styledComponents: true,
  pageExtensions: ['page.tsx', 'page.ts', 'page.jsx', 'page.js', 'api.ts', 'api.js']
}

module.exports = nextConfig
```

Figure 4.2: *Required changes in configuration to use framework as REST API*

We will add **`api.js/ts`** requirements into the configuration. This will help us to create a more beautiful URL for the API as shown in *Figure 4.3:*

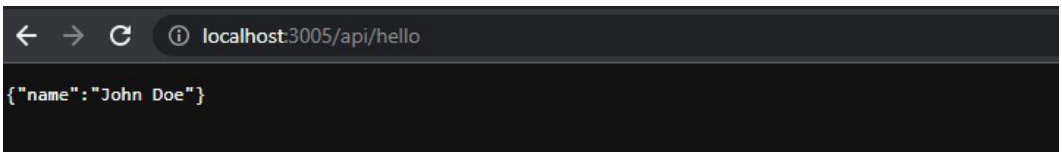


Figure 4.3: *Response of the api using new configuration*

Creating the simple NextJS API routing structure

In the previous chapters, we were making the authorization form that will require an API call that will allow the user to enter the private area. We will stick to this task when we make a simple routing structure.

As all our API endpoint files should be located in the **API** folder let us create the **auth** folder inside and create several endpoints as shown in *Figure 4.4:*

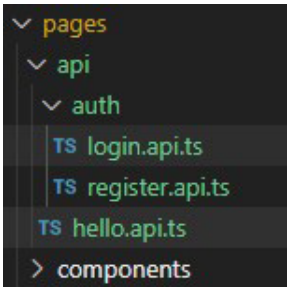


Figure 4.4: *API folder structure that we will use for the application*

We added 2 endpoint files. Inside **hello.api.ts** file place code from *Figure 4.5* to make these endpoints work:

```
import type { NextApiRequest, NextApiResponse } from 'next'

type Data = {
    login: string
}

export default function handler(
    req: NextApiRequest,
    res: NextApiResponse<Data>
) {
    res.status(200).json({ login: 'John Doe' })
}
```

Figure 4.5: *Code source for the api endpoint files*

Now when we will try to enter the endpoint in the browser (or any application to check REST requests like Postman) we will see this in the response (*Figure 4.6)*:

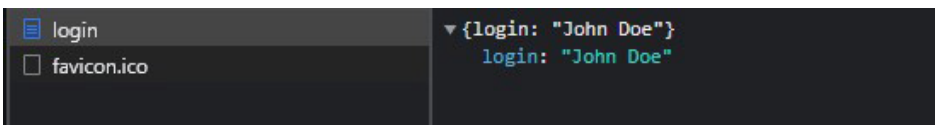


Figure *4.6: Response for the login url call*

Creating the simple NextJS REST API

To have a well-architecture REST API, we should remember that all our calls should be separated to have **`GET`**, **`POST`**, **`UPDATE`** and **`DELETE`** requests. The type of request will come in the handler function parameter **`req.`** To use it we can call **`req.method`** like in the following figure:

```
export default function handler(req, res) {
  if (req.method === 'GET') {
    // Process a GET request
  } else {
    // Handle any other HTTP method
  }
}
```

Figure 4.7: *Example of how to separate different types of HTTP call*

Let us stick to the login endpoint. For now, we will use only the POST type to send login and password into the API and get the result from it that will contain rejection or the authorization key that we will use in further requests. We will use the same way as we did for the frontend part and create the **`core`** folder that will contain business logic for the API. We will need it to make the API at the NextJS side more abstract to have the possibility to use the API as backend-for-fronted and as a stand-alone API with the same code base.

Now to create our first simple API endpoint let us follow these steps. Create the configuration file in **`api/core/`** folder and name it **`configuration.ts.`** Put this code into it. We will need the enum with default messages for our API in the following figure:

```
enum Configuration {
    CORRECT_REQUEST = 'Success !',
    WRONG_REQUEST = 'No such request'
}

export { Configuration }
```

Figure 4.8: *API configuration file source*

After that, we can add this code to the **`login.api.ts`** file in the following figure:

```
import type { NextApiRequest, NextApiResponse } from 'next'
import { Configuration } from '../core/configuration'

type Data = {
    message: string
}

export default function handler(
    req: NextApiRequest,
    res: NextApiResponse<Data>
) {
    if (req.method === 'POST') {
        res.status(200).json({ message: Configuration.CORRECT_REQUEST })
    } else {
        res.status(200).json({ message: Configuration.WRONG_REQUEST })
    }
}
```

Figure 4.9: *Login endpoint code*

To test all REST requests we will use the Postman app (you can use any that is more comfortable for you). This is what will be for the **`POST`** request shown in the following *figure 4.10*:

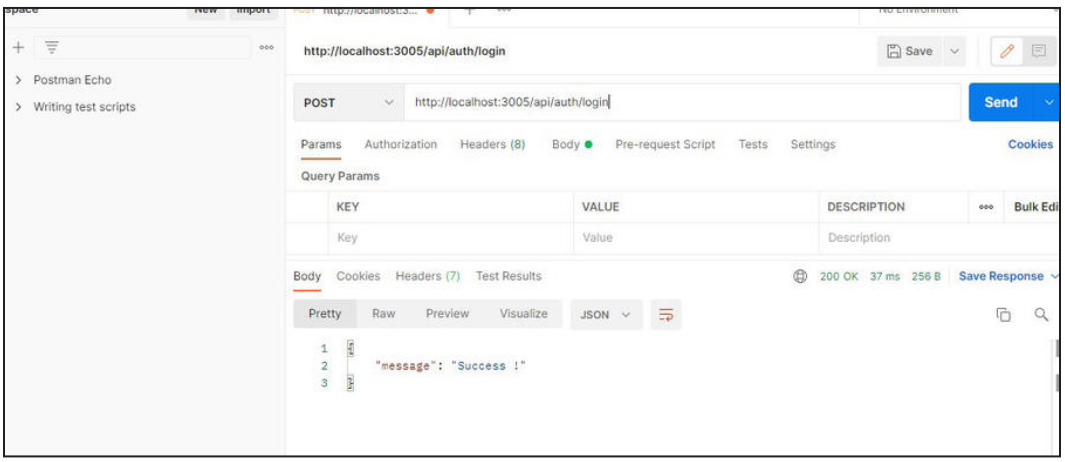


Figure 4.10: *Test of the API call in Postman application for the POST request*

That is what we will see for the **`GET`** request shown in the following *Figure 4.11:*

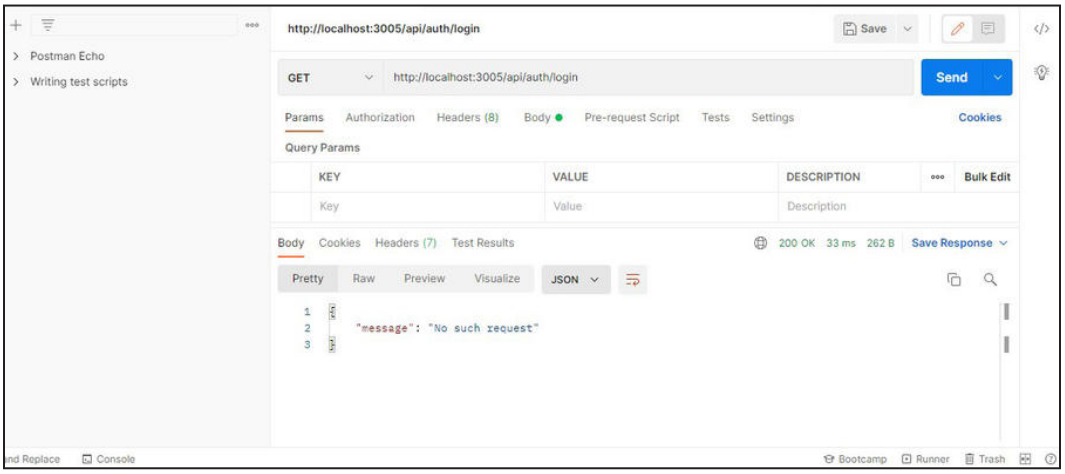


Figure 4.11: *Test of the API call in Postman application for the GET request*

The **`"No such request"`** message that we provide from the configuration.

Generating an authorization token for the user

For the authorization API, we will need the token key that will be used whenever we will need to get something from the API inside the personal user area.

To complete this task we will need a simple string generation function. In real-life applications, we would need to encode and prepare with some business logic the token. We will not stick to it as it is not part of our topic. For now, we need some code that will be unique. For this let us create the utils file where we could store functions like this shown in the following *Figure 4.12:*

```
const generateToken = () => {
    return Math.random().toString(36).substr(2) + Math.floor(Date.now() / 1000)
}

export { generateToken }
```

Figure 4.12: *Function for the random token key generation*

We can try this function as shown in *Figure 4.13:*

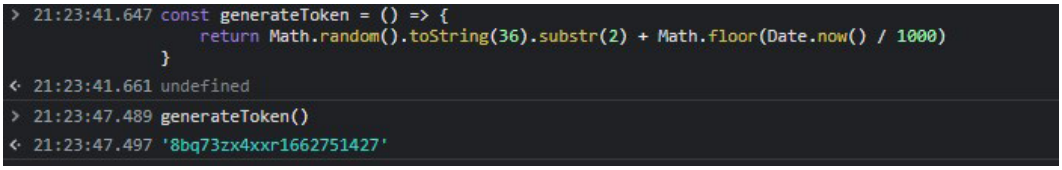


Figure 4.13: *Result of generateToken function call in browser console*

As we can see - now we have a random string token that we could use for the login API.

Using Singleton, Builder, and Strategy patterns in API construction

As we learned from previous chapters, using the patterns will help us create readable and maintainable architecture that we can change or scale by request. So let us speak about where to start and what design we will use for each action and property.

The list of patterns by the requirements will look like this:

- Singleton for login request and login state
- Strategy for the request type (as we will need a possibility to connect to an external API)
- Builder for the user profile build

Baking Singleton for API

We will start with file creation and the name of the file will be **`login.service.ts`**. This file will contain the login call as a function and login state. We will also make the condition private so we will need the getter function. (*Figure 4.14*):

```
class LoginService {
    private static instance: LoginService;
    private isLoggedIn: boolean = false;
    private constructor() {}

    public static getInstance(): LoginService {
        if (!LoginService.instance) {
            console.log('LoginService new instance')
            LoginService.instance = new LoginService();
        }
        return LoginService.instance;
    }

    login() {
        // Here we will provide the login logic depending on what strategy is selected
        this.isLoggedIn = true;
    }

    getLoginStatus() {
        return this.isLoggedIn
    }
}

export { LoginService }
```

Figure 4.14: *Login singleton code*

For now, we do not have any business logic for login as we will need to make the login itself in the subsequent implementation with Strategies. After that, we will come back and add this logic to our Singleton.

Baking Strategy for API

As we declared before a strategy pattern is required to separate possible ways of getting the data from the data source and keep the logic structure the same. Let us create the file with the name **`login-strategy.ts.`** Inside we will develop the strategy class that will be called LoginContext with the login method inside. As we need to use the same structure we will create the interface ILoginStrategy also. We will require to make the mocks as we did it before also.

Let us start coding. First, we will create the mock data. To complete we will follow the steps:

1. Create the file with mock data in the folder where we store mock data for the application.(**`<root>/pages/mocks`**)
2. Create file **`users.json`** and fill it with the data from *Figure 4.15:*

```
[
    {
        "user": "testUser",
        "password": "asdqwe123"
    }
]
```

Figure 4.15: *Mock data for the users*

3. After that we can connect mock data to the application as its presented on *Figure 4.16:*

```
import articles from "./articles.json"
import baking from "./baking.json"
import users from "./users.json"

const getMock = {
    articles,
    baking,
    users
}

export { getMock }
```

Figure 4.16: *Connection of the mocks to mocking system*

4. Finally, we can create the strategy context file that will use strategies for the login. To achieve this please create the **login-strategy.ts** file in **`<root>/pages/api/core`** folder and fill it with code from *Figure 4.17:*

```
class LoginContext {
    private strategy: ILoginStrategy;

    constructor(strategy: ILoginStrategy) {
        console.log('Login strategy class is', strategy)
        this.strategy = strategy
    }

    public useLogin(user: string, password: string): IUser {
        console.log('Now login is on fire')
        return this.strategy.login(user, password)
    }
}
```

***Figure 4.17**: Context file source that presents a strategy context*

This is what the strategy context will look like. We will use a defined strategy class on construction calls. For our case we do not have any data source except local mocks so we will require this concrete strategy: (*Figure 4.18):*

```
class LoginWithMock implements ILoginStrategy {
    public login(user: string, password: string) {
        const users = getMock.users
        const checkUser = users.find((userItem: {user: string, password: string}) => {
            return userItem.user === user && userItem.password === password
        })
        let loginState = { state: false, token: '' }
        if (checkUser) {
            loginState = { state: true, token: generateToken() }
        }
        return loginState;
    }
}
```

***Figure 4.18:** This is an exact strategy class code that will be used for the API*

Now we can update the service but before that do not forget to add the required interfaces to not use any in this case as shown in *Figure 4.19:*

```ts
interface IUser {
    state: boolean;
    token: string;
}

interface ILoginStrategy {
    login(user: string, password: string): IUser;
}
```

***Figure 4.19**: Interfaces that will be used as a class types*

To update the service we need to change **`LoginService`** class and add token param like the following figure:

```ts
import { loginType } from "./configuration";
import { LoginContext } from "./login-strategy";
import { IUser } from "./types";

class LoginService {
    private static instance: LoginService;
    private isLoggedIn: boolean = false;
    private token: string = '';
    private loginState: IUser | null = null;
    private constructor() {}

    public static getInstance(): LoginService {
        if (!LoginService.instance) {
            console.log('LoginService new instance')
            LoginService.instance = new LoginService();
        }
        return LoginService.instance;
    }

    async login(user: string, password: string) {
        // Here we will provide the login logic depending on what strategy is selected
        const loginContext = new LoginContext(loginType);
        this.loginState = await loginContext.useLogin(user, password)
        this.isLoggedIn = this.loginState && this.loginState.isLoggedIn;
        this.token = this.loginState.token;
    }

    getLoginStatus() {
        return this.isLoggedIn
    }

    getToken() {
        return this.token
    }

}

export { LoginService }
```

***Figure 4.20:** Updated code for the login singleton*

Also, let us do some changes in the configuration file to have the login type scripted like the following figure:

```
import { LoginWithGQL, LoginWithMock, LoginWithAmplify } from "./login-strategy"

enum Configuration {
    CORRECT_REQUEST = 'Success !',
    WRONG_REQUEST = 'No such request'
}
enum LoginStrategiesNames {
    MOCK = 'mock'
}

const LoginStrategies = {
    [LoginStrategiesNames.MOCK]: new LoginWithMock()
}

const currentLoginStrategy = LoginStrategies[LoginStrategiesNames.AMPLIFY]

export { Configuration, currentLoginStrategy, UserBuilderMethods }
```

***Figure 4.21**: Updated configuration for the login*

Perfect. We can now implement this service into the login component we made before. But before we switch to implementation let us proceed with the builder.

Baking Builder for API

As we planned before we will use Builder for the user account data as there could be different types of users, permissions, and so on. Let us just wrap it with a builder pattern to make it easier to scale if something will be changed.

Inside the builder, there will be several producers that will be responsible for each type of user that we could want to create at the current state, as shown in *Figure 2.22:*

```
import { UserBuilderMethods } from "./configuration";
import { IUser } from "./types";

interface UserBulder {
    [UserBuilderMethods.PRODUCE_REGULAR_USER](): void;
    [UserBuilderMethods.PRODUCE_UPDATED_USER](): void;
    [UserBuilderMethods.PRODUCE_ADMIN_USER](): void;
}

class ApplicationUser implements UserBulder {
    constructor(private user: IUser) {}

    [UserBuilderMethods.PRODUCE_REGULAR_USER](): void {
        console.log('trigger build')
        this.user.userPropertiesActions?.push('Regular properties')
    }
    [UserBuilderMethods.PRODUCE_UPDATED_USER](): void {
        this.user.userPropertiesActions?.push('Updated properties')
    }
    [UserBuilderMethods.PRODUCE_ADMIN_USER](): void {
        this.user.userPropertiesActions?.push('Admin properties')
    }
}

export { ApplicationUser }
```

Figure 4.22: *Builder class realization*

As you can see we use the same code style as before so do not forget to create all constants in the configuration as illustrated in *Figure 4.23:*

```
import { LoginWithMock } from "./login-strategy"

enum Configuration {
    CORRECT_REQUEST = 'Success !',
    WRONG_REQUEST = 'No such request'
}
enum LoginStrategiesNames {
    MOCK = 'mock'
}

enum UserBuilderMethods {
    PRODUCE_REGULAR_USER = 'produceRegularUser',
    PRODUCE_UPDATED_USER = 'produceUpdatedUser',
    PRODUCE_ADMIN_USER = 'produceAdminUser'
}

const LoginStrategies = {
    [LoginStrategiesNames.MOCK]: new LoginWithMock(),
}

const loginType = LoginStrategies[LoginStrategiesNames.MOCK]

export { Configuration, loginType, UserBuilderMethods }
```

Figure 4.23: *Updated configuration to optimize builder code*

We will also require changes in the data source as we now expect more data from the source as illustrated in *Figure 4.24:*

```
[
    {
        "user": "testUser",
        "password": "asdqwe123",
        "userProperties": ["PRODUCE_REGULAR_USER"]
    }
]
```

Figure 4.24: *Update in the users.json file*

Now we need to make the last modifications in the service to get the builder into the working state as shown in *Figure 4.25:*

```
import { types } from "sass";
import { loginType, UserBuilderMethods } from "./configuration";
import { LoginContext } from "./login-strategy";
import { ApplicationUser } from "./user-builder";

class LoginService {
    private static instance: LoginService;
    private isLoggedIn: boolean = false;
    private token: string = '';
    private applicationUser: ApplicationUser = {} as ApplicationUser
    private constructor() {}

    public static getInstance(): LoginService {
        if (!LoginService.instance) {
            console.log('LoginService new instance')
            LoginService.instance = new LoginService();
        }
        return LoginService.instance;
    }

    login(user: string, password: string) {
        // Here we will provide the login logic depending on what strategy is selected
        const loginContext = new LoginContext(loginType);
        const loginState = loginContext.useLogin(user, password)
        console.log('loginState', loginState)
        this.isLoggedIn = loginState.state;
        this.applicationUser = new ApplicationUser(loginState)
        loginState.userProperties.forEach((property: keyof typeof UserBuilderMethods) => {
            UserBuilderMethods[property] && this.applicationUser[UserBuilderMethods[property]]()
        })
        this.token = loginState.token;
    }

    getLoginStatus() {
        return this.isLoggedIn
    }

    getToken() {
        return this.token
    }
}

export { LoginService }
```

Figure 4.25: *Singleton update to properly login user*

As you can see login method now operate with builder methods and create all required data inside of it.

Let us switch to the UI form. Inside of it, we will need to get the data from the inputs and send them into the service. So, your login form component should look like as illustrated in *Figure 4.26:*

```tsx
import TextInput from '../atoms/TextInput'
import PasswordInput from '../atoms/PasswordInput'
import SubmitButton from '../atoms/SubmitButton'
import { TestIDs } from '../../pages/core/configs'
import styles from '../../styles/LoginForm.module.scss'
import { LoginService } from '../../pages/api/core/login.service'
import { FormEvent, useState } from 'react'
import { useAppDispatch } from '../../pages/hooks'
import { changeAuthState } from "../../pages/store/authSlice";
import { useRouter } from 'next/router'

const LoginForm = () => {
    const [login, setLogin] = useState('');
    const [password, setPassword] = useState('');
    const router = useRouter();

    const dispatch = useAppDispatch();

    const errorTestID = TestIDs.ERROR
    const loginService = LoginService.getInstance()
    const loginAction = async (event: FormEvent<HTMLFormElement>) => {
        console.log('login');
        event.preventDefault();
        loginService.login(login, password);
    }

    const loginEnter = (value: string ) => {
        setLogin(value);
    }

    const passwordEnter = (value: string ) => {
        setPassword(value);
    }

    return (
        <section className={styles.LoginForm}>
            <form onSubmit={loginAction}>
                <div>
                    <TextInput id="login" onLoginEnter={loginEnter} />
                </div>
                <div>
                    <PasswordInput id="password" onPasswordEnter={passwordEnter} />
                </div>
                <div>
                    <SubmitButton id="submit-login" />
                </div>
                <div data-testid={errorTestID} />
            </form>
        </section>
    )
}

export default LoginForm
```

Figure 4.26: *Login for component code*

Now when we try to log in with credentials, we will fill the service data singleton. That means that we could use this data on each page (if we do not reload the page). To solve this problem, we will use one of the ways to store data in the browser but for now it is enough to have the data itself.

To check that data is still in instance update the login page with the link to the main page will appear as *Figure 4.27*:

```
import type { NextPage } from 'next'
import Link from 'next/link'
import styles from '../styles/Login.module.scss'
import LoginForm from '../ui/molecules/LoginForm'

const LoginPage: NextPage = () => {
  return (
    <article>
      <LoginForm />
      <Link href="/"><a className={styles.link}>Main</a></Link>
    </article>
  )
}

export default LoginPage
```

Figure 4.27: *Login page component*

Now on router change inside your console, you will see that the login service still contains user data. That means that we can use this data for any purpose inside the application as shown in the following figure:

```
user                                                    index.page.tsx?ac9a:9
▼LoginService {isLoggedIn: true, token: 'zh978ggt30k1663073023', applicationUser: ApplicationUser} ⓘ
  ▼applicationUser: ApplicationUser
    ▼user:
       state: true
       token: "zh978ggt30k1663073023"
      ▼userProperties: Array(1)
         0: "PRODUCE_REGULAR_USER"
         length: 1
        ▶[[Prototype]]: Array(0)
      ▶[[Prototype]]: Object
    ▶[[Prototype]]: Object
    isLoggedIn: true
    token: "zh978ggt30k1663073023"
  ▶[[Prototype]]: Object
```

Figure 4.28: *Result of the login response in the browser console*

Using the Apollo client for NextJS

To start using the client we need to do a short setup to activate the server that we could use for the requests to get the data from the GraphQL requests.

Apollo Server is working the same way as any regular node server software (for example : Express or similar). To start setup we will need to create a folder and install it. We recommend doing it in the same project folder for education purposes and in a separate folder for real-world applications.

Enter the command as illustrated in *Figure 4.29:* in your console to create the required folder:

Figure 4.29: *Commands to create the folder for the GraphQL server*

After that we need to **`init npm`** project by entering the command shown in *Figure 4.30:*

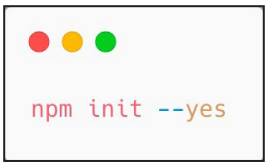


Figure 4.30*: Command to init the application*

Now we can add required dependencies to our server using npm as illustrated in *Figure 4.31*:

Figure 4.31: *Command to install the server in folder as package*

Creating the model for the NextJS application

As for now, we need only users we will create the Scheme in *Figure 4.32* this to operate with users:

```
type User {
  user: String!
  password: String!
  userProperties: [String!]!
}

type Query {
  getUser(user: String!, password: String!): User
}
```

Figure 4.32: *GraphQL schemas for the users*

Writing the connecting system for Apollo

Let us create the server index file (like we do for any node js server). Create the index.js file in the server project root (use CLI or your IDE for it). Inside this file, we can create the schema and queries to get the data.

```
const { ApolloServer, gql } = require('apollo-server');
const users = require('../pages/mocks/users.json')

const typeDefs = gql`
  type User {
    user: String!
    password: String!
    userProperties: [String!]!
  }

  type Query {
    getUser(user: String!, password: String!): User
  }
`;

const resolvers = {
    Query: {
        getUser: (obj, params) => {
            return users.find(user => user.user === params.user && user.password ===
params.password)
        },
  };

const {
    ApolloServerPluginLandingPageLocalDefault
  } = require('apollo-server-core');

  const server = new ApolloServer({
    typeDefs,
    resolvers,
    csrfPrevention: true,
    cache: 'bounded',
    plugins: [
      ApolloServerPluginLandingPageLocalDefault({ embed: true }),
    ],
  });

server.listen().then(({ url }) => {
console.log(`🚀  Server ready at ${url}`);
});
```

Figure 4.33: *Server file source to start the Apollo server locally*

To be more specific here we will go through this file together step by step:

1. **const users**: here we are getting data from mocks that were created before.
2. **const typedef**: this is a GraphQL Scheme that we will use in the application.
3. **const resolvers**: this is a definition of actions that will be triggered by query call from the Scheme.
4. **const server**: this is a server instance where we connect all together .
5. **server.listen**: is a function that starts the server instance.

Now after the application start (type: **`node index.js`** in your console) we will see as shown in the following figure:

```
🚀  Server ready at http://localhost:4000/
```

Figure 4.34: *Success response log in the console after server start*

To check that server is up and running we will open a sandbox here **https://studio.apollographql.com/sandbox** and follow the instructions to add your current local host into the sandbox like in *Figure 4.35:*

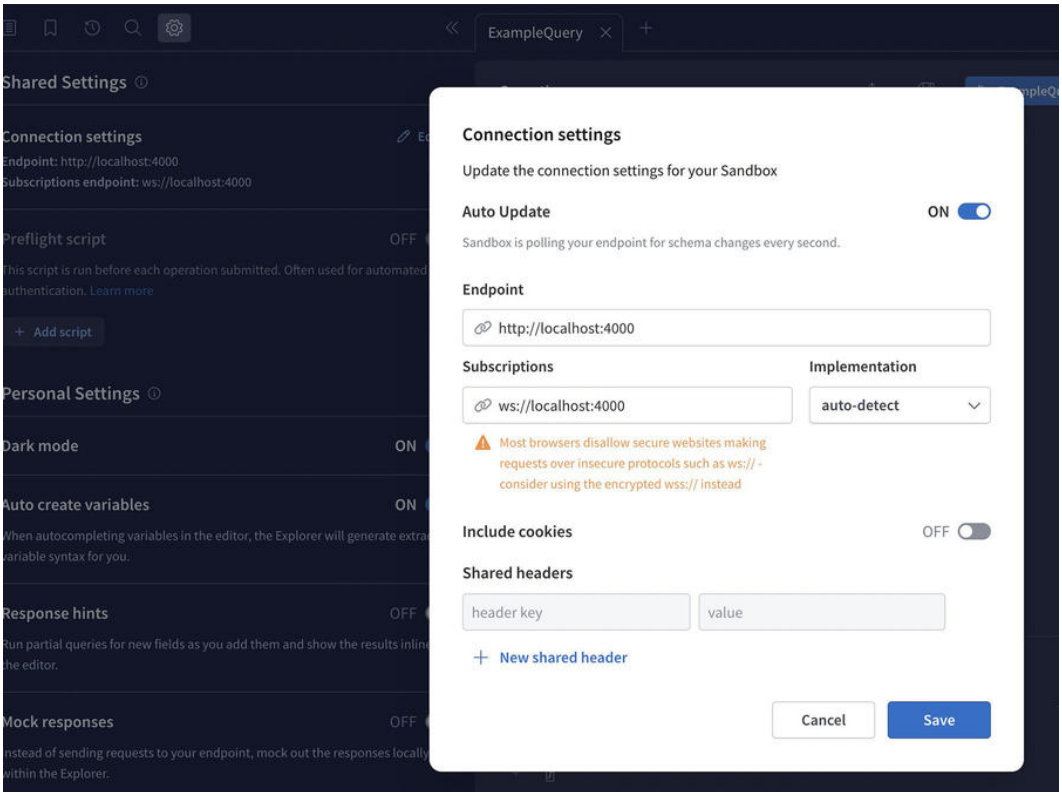


Figure 4.35: *Sandbox page that we will use for the testing*

After that we can execute our first query by adding the following data into the query frame of the page *(Figure 4.36):*

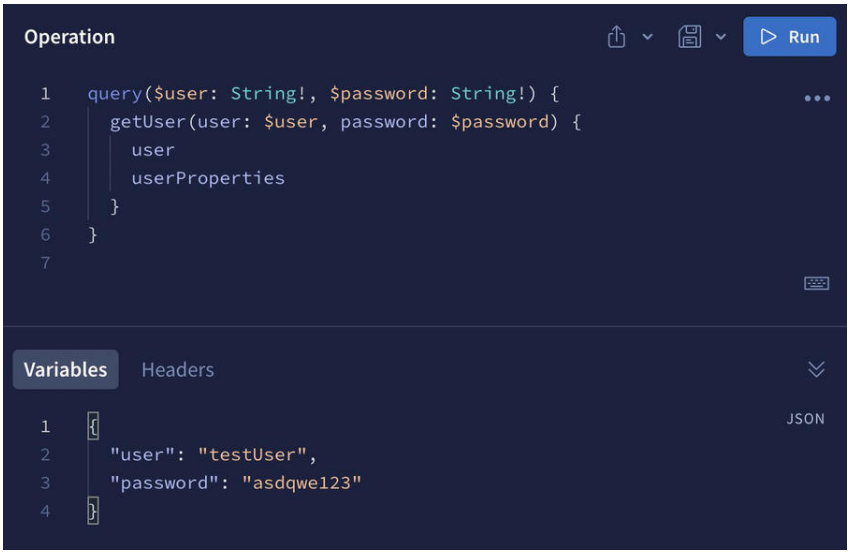


Figure 4.36: *Query frame in the sandbox*

Now on pressing the **Run** button at the right top corner we will see the result as shown in *Figure 4.37:*

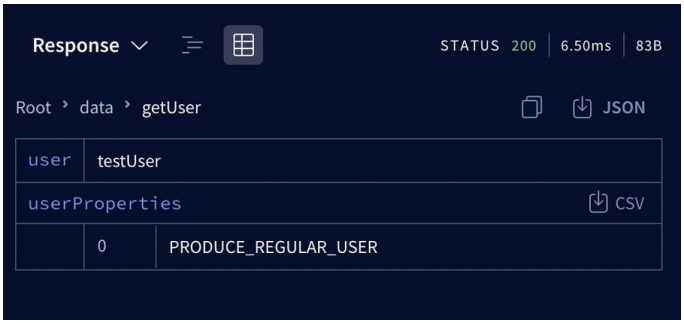


Figure 4.37: *Result frame in the sandbox*

As you can see we got the correct user data.

Keeping in mind that we are making basic login and in the real application you should not keep a real password in the database as should never send a real, not encrypted password as a parameter from the form. Please check specific documentation about this topic. For this book, this subject is out of scope.

Reusing API from the previous recipe for Apollo

As you remember in the previous part we made a strategy pattern and we made it for the purpose. So now to reuse the same logic we need to add a new strategy class, with only one update that would be required. As we will use GraphQL request we would need to add **async/await** to methods that are already in the system.

First, we need to add Promise to the login method in the login strategy interface as illustrated in *Figure 4.38*:

```
interface ILoginStrategy {
    login(user: string, password: string): Promise<IUser>;
}
```

Figure 4.38: *Updates that will be used in login strategy class*

Then we will need some updates in the login service as illustrated in *Figure 4.39*:

```
// add async for the funciton description
async login(user: string, password: string) {
        // Herewe will provide the login logic depending on what strategy is selected
        const loginContext = new LoginContext(loginType);

        // add await for this function call
        const loginState = await loginContext.useLogin(user, password)

        this.isLoggedIn = loginState.state;
        this.applicationUser = new ApplicationUser(loginState)
        loginState.userProperties.forEach((property: keyof typeof UserBuilderMethods) => {
            UserBuilderMethods[property] && this.applicationUser[UserBuilderMethods[property]]()
        })
        this.token = loginState.token;
    }
```

Figure 4.39: *Adding the async/await functionality for the login method*

Now all login flow will be asynchronous and we can add the new strategy using the Apollo client as illustrated in *Figure 4.40*:

```
class LoginWithGQL implements ILoginStrategy {
    async gqlLogin(user: string, password: string) {
        const { data } = await client.query({
            query: gql`
              query {
                getUser(user: "${user}", password: "${password}") {
                  user
                  userProperties
                }
              }
            `,
          });
        return await data
    }
    public async login(user: string, password: string) {
        let loginState = { state: false, token: '', userProperties: [] }
        const checkUser = await this.gqlLogin(user, password)
        if (checkUser && checkUser.getUser) {
            loginState = { state: true, token: generateToken(), userProperties: checkUser.getUser.userProperties }
        }
        return loginState
    }
}
```

Figure 4.40: *Login with GraphQL strategy class*

Having this class, we can update the configuration for the API like this to activate the new strategy as illustrated in *Figure 4.41*:

```
enum LoginStrategiesNames {
    MOCK = 'mock',
    GQL = 'gql'
}

const LoginStrategies = {
    [LoginStrategiesNames.MOCK]: new LoginWithMock(),
    [LoginStrategiesNames.GQL]: new LoginWithGQL(),
}

const loginType = LoginStrategies[LoginStrategiesNames.GQL]
```

Figure 4.41: *Configuration update for the GraphQL realization*

As you can see we have a minimal update in the code. Also if we will need to add a new strategy it will require a minimum of updates in the code.

Setting up an Apollo client for NextJS

To add the Apollo client to your application we will first need to add it to our project similar to what we did for the server as shown in *Figure 4.42:*

```
// if you use yarn
yarn add @apollo/client graphql

// if you use NPM
npm install @apollo/client graphql
```

Figure 4.42: *Commands to add Apollo client to the project*

When we successfully added the Apollo to our dependencies we can create the client that we will use in the application. To do it please create the file named **`apollo-client.js`** with the following code:

```
import { ApolloClient, InMemoryCache } from "@apollo/client";

const client = new ApolloClient({
    uri: "http://localhost:4000",
    cache: new InMemoryCache(),
});

export default client;
```

Figure 4.43: *Source of the apollo-client.js file*

The URL in Apollo Client object parameters is the URL of the server that we created before. If you deployed it yourself (or had a server before) please use this URL in this configuration.

Now we can open the login form page and try to log in again with the same credentials. And console should return the same information as before as shown in *Figure 4.44:*

```
login                                                    LoginForm.tsx?81f1:16
Login strategy class is ▸ LoginWithGQL {}                login-strategy.ts?5011:53
Now login is on fire                                     login-strategy.ts?5011:58
data ▸ {getUser: {…}}                                    login-strategy.ts?5011:34
checkUser ▸ {getUser: {…}}                               login-strategy.ts?5011:40
loginState                                               login.service.ts?d5fa:25
▾{state: true, token: '6hqvtvpin451663251554', userProperties: Array(1)} i
   state: true
   token: "6hqvtvpin451663251554"
  ▾userProperties: Array(1)
     0: "PRODUCE_REGULAR_USER"
     length: 1
    ▸[[Prototype]]: Array(0)
  ▸[[Prototype]]: Object
trigger build                                            user-builder.ts?2efc:14
```

Figure 4.44: *Result of the api call in the browser console*

Conclusion

In this chapter, we have been introduced to the server-side potential of NextJS. This knowledge will allow us to create any kind of API using NextJS only no matter what purpose we need. As you can see the server side of the NextJS is similar to a regular NodeJS application and can reuse any logic from your Express application for example. Also now we can choose between the type of the API, REST or GrapthQL.

In the next chapters, we will do research in the state management area to make using any data on the client side more smooth. We will also be introduced to AWS Amplify for the API creation where we could use the knowledge from this chapter about GraphQL and REST.

Chapter 5
Using state management in NextJS

Introduction

Sooner or later the application will start growing, ant number of components also. As a developers we would like to have a system to communicate between components faster and in the same time have a possibility to debug the data that is passing from one component to another. For these purposes we will need a state management system that we will discover in this chapter.

Structure

- Using state-management tools in applications
- Setting up Redux in NextJS
- Writing tests for the store before we start coding
- Creating Redux store objects in NextJS
- Using the store for authorization in our application
- Connecting data API to state management
- Conclusion

Objectives

This chapter will teach how to add and use the state management system. As an example system, we will use Redux as the most efficient and popular state management system for ReactJS. Also, we will walk through the flow of creating a store and code tests for the store and connect the store to the API from the previous chapter.

Using state-management tools in applications

The architecture of web applications is not only in data structures and data design patterns. We also need to think about how to operate data between components and react to data changes in these components. We need to create the mechanism that will make the state of any object and follow the state machine principles.

In the web application world, we have several libraries for different platforms. As NextJS is based on React we will use Redux as a state management system.

In our application, we will need it to share the login information. Also, we will add the kitchen system to order and cook the burgers we made in previous chapters.

Setting up Redux in NextJS

Redux is based on three principles:

- Single source of truth: The state of an entire application is stored in a single object tree within a single store.
- State is read-only: The only way to change the state is by dispatching an action, which is a plain JavaScript object describing what happened.
- Changes are made with pure functions: To specify how the state tree is transformed by actions, you write pure reducers.

To add Redux to our project type following commands in the folder root: (*Figure 5.1*)

```
// If you use yarn
yarn add @reduxjs/toolkit react-redux
yarn add next-redux-wrapper

// If you use npm
npm install @reduxjs/toolkit react-redux
npm install next-redux-wrapper
```

Figure 5.1: *Commands to add Redux to the project*

Writing tests for the store before we start coding

As you remember, we use the test driven approach in this book to create an application and any modules for the application. Implementing Redux will not be an exclusion, but we have to introduce some things that will help you to more clearly understand the whole way of using Test-driven development with Redux.

For the start, we will require to make some changes in the folder structure of the application. Please add the **`store folder`** to the **`pages`** folder. Inside the store folder, we will require **`__tests__`** folder as well to collect the tests for the Redux store as in *Figure 5.2:*

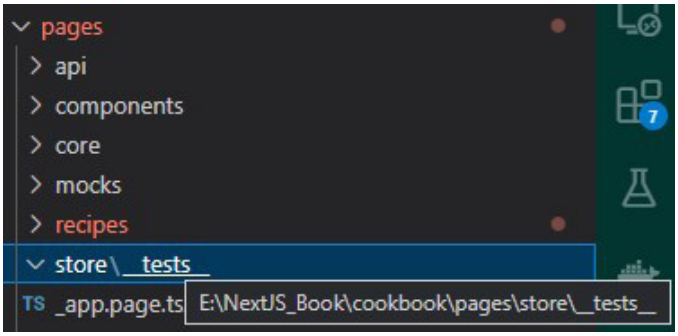


Figure 5.2: *Updated folder structure to collect Redux files*

Inside these folders, we will create our first files that will correspond to the auth state for the application, like in *Figure 5.3:*

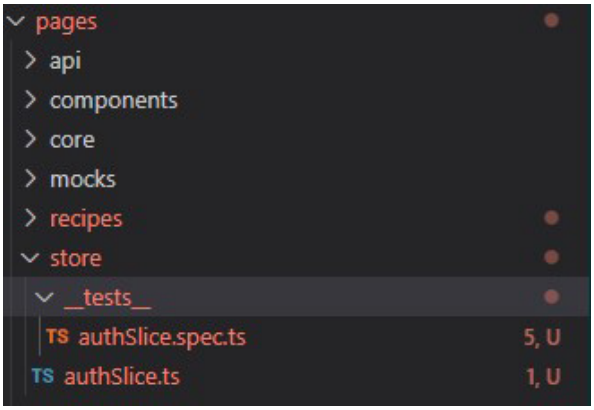


Figure 5.3: *Files for the auth state*

In the specification file, we will add this code to start creating the tests. For now, it will pass but we will remove the rule for test in the next steps: (*Figure 5.4*)

```
describe("Auth Slice", () => {
  describe("My First function", () => {
    it("should compile", () => {
      expect(1).toEqual(1);
    });
  });
});
```

Figure 5.4: *The code from* ***authSlice.spec.ts*** *file*

Also, we will require to fill the store file with the code illustrated in *Figure 5.5:*

```
import { createSlice } from "@reduxjs/toolkit";
const authSlice = createSlice({
  name: "auth",
  reducers: {},
  initialState: {},
});
export default authSlice.reducer;
```

Figure 5.5: *Code for store file*

We will add enough code to compile but stick to the rule that complete code should be written after the test file is ready.

We will add the code into the store file to have something compile like *Figure 5.6:*

```
import { createSlice } from "@reduxjs/toolkit";

type Auth = {
    isLoggedIn: boolean;
}

export type AuthState = {
    auth: Auth
};

export const INITIAL_STATE: AuthState = {
    auth: {
        isLoggedIn: false
    },
};

const authSlice = createSlice({
  name: "auth",
  reducers: {},
  initialState: INITIAL_STATE,
});

export default authSlice.reducer;
```

Figure 5.6: *Minimal code to compile the store*

We need to modify the reducers part to have a dummy reducer that will change the state of the auth like *Figure 5.7:*

```
const authSlice = createSlice({
  name: "auth",
  reducers: {
    changeAuthState: (state: RootState, action: PayloadAction<string>) => {
        return state;
    }
  },
  initialState: INITIAL_STATE,
});

export const { changeAuthState } = authSlice.actions;
```

Figure 5.7: *Code update for the reducer*

Now we can create the first test that will fail (in our case because we will make it fail for now). The code is described in *Figure 5.8:*

```
import authSlice, {
    changeAuthState,
    INITIAL_STATE,
    Auth,
    AuthState
} from '../authSlice';

describe("Auth Slice", () => {
  describe("My First function", () => {
    it("should auth the user in the store", () => {
        const auth: Auth = {
            isLoggedIn: true
          };
          const action = changeAuthState(auth);
          const expectedResult: AuthState = {
            auth,
          };
          const actualResult = authSlice(INITIAL_STATE, action);
          expect(actualResult).toEqual(expectedResult);
    });
  });
});
```

***Figure 5.8**: Test file code, that will be failed for now*

Now on the test start, we will get the message about failing the test like in *Figure 5.9*:

```
  Object {
    "auth": Object {
-     "isLoggedIn": true,
+     "isLoggedIn": false,
    },
  }

  17 |           };
  18 |           const actualResult = authSlice(INITIAL_STATE, action);
> 19 |           expect(actualResult).toEqual(expectedResult);
     |                                ^
  20 |     });
  21 |   });
  22 | });

  at Object.toEqual (pages/store/__tests__/authSlice.spec.ts:19:32)
```

***Figure 5.9**: Failing test message*

To pass the test we need to modify the reducer to change the state with an action. Please change the reducer code with the code provided in *Figure 5.10*:

```
const authSlice = createSlice({
  name: "auth",
  reducers: {
    changeAuthState: (state: AuthState, action: PayloadAction<Auth>) => {
        const newAuth = action.payload;
        state.auth = newAuth;
      }
  },
  initialState: INITIAL_STATE,
});
```

Figure 5.10: *Updated reducer to pass the test*

Now when we try to start the test again we will get this result: (*Figure 5.11*)

```
$ jest
 PASS  __tests__/ui.test.tsx
  ● Console

    console.log
      LoginService new instance

      at Function.log [as getInstance] (pages/api/core/login.service.ts:15:21)

 PASS  __tests__/index.test.jsx
  ● Console

    console.log
      LoginService new instance

      at Function.log [as getInstance] (pages/api/core/login.service.ts:15:21)

    console.log
      user LoginService { isLoggedIn: false, token: '', applicationUser: {} }

      at log (pages/index.page.tsx:9:11)

 PASS  pages/store/__tests__/authSlice.spec.ts

Test Suites: 3 passed, 3 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        4.282 s
Ran all test suites.
Done in 6.02s.
```

Figure 5.11*: All tests are passed and green*

Creating Redux store objects in NextJS

To connect the store to our application we will need to create some objects. To achieve it we will make some updates to the file structure and application files.

First, we will create the **`index.ts`** file in the store folder root like *Figure 5.12*.

***Figure 5.12**: Index file in the store folder root*

Add the code from *Figure 5.13* inside this file:

```
import { configureStore } from '@reduxjs/toolkit'
import authSlice from './authSlice';
import { createWrapper } from "next-redux-wrapper";
import articleSlice from './articleSlice';

export const store = configureStore({
  reducer: {
    [authSlice.name]: authSlice.reducer,
  },
  devTools: true,
});


const makeStore = () => store;

export type RootState = ReturnType<typeof store.getState>
export type AppDispatch = typeof store.dispatch
export const wrapper = createWrapper<RootState>(makeStore);
```

***Figure 5.13:** Source of the index file*

Let me explain a little about what we have inside this file. First, we need to configure the store itself and we will use the configure store function for it. As you remember we created the object in the auto slice that contains the name, actions and reducers. So, we will use the name as a string and reducer as the reducer for the store. Also, we will add a property devTools to get the information in the special browser extension.

One line can be confusing it is the line from *Figure 5.14:*

```
const makeStore = () => store;
```

Figure 5.14: *Strange part of the file*

This **`makeStore`** function is required by the wrapper function for the NextJS. This code is only needed if you use the framework. In regular React applications, it will not be required. So just copy and paste it from the example.

Next what we need to add the possibility to use hooks for the store. To add this possibility to the application we will create a folder with the index file inside. It is not part of the store as we could create and add hooks unrelated to the Redux.

Please check *Figure 5.15* to make changes in your folder structure:

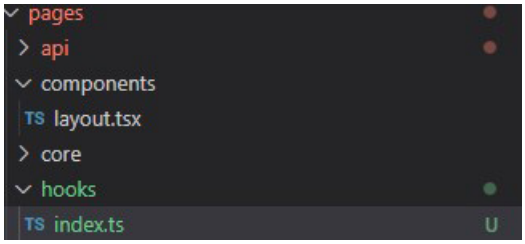


Figure 5.15: *Hooks folder and index file*

Inside the index file, we will add this code to make hooks work for the application: (*Figure 5.16)*

```
import { useDispatch, useSelector } from 'react-redux'
import type { TypedUseSelectorHook } from 'react-redux'
import type { RootState, AppDispatch } from '../store'

export const useAppDispatch: () => AppDispatch = useDispatch
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector
```

Figure 5.16: *Hooks file source code*

Now we are ready to add the store to the application. Let us do some changes in the app file of the application like in *Figure 5.17:*

```
import '../styles/globals.scss'
import { Provider } from "react-redux";
import type { AppProps } from 'next/app'
import Layout from './components/layout'
import { wrapper } from "./store";

function CookBook({ Component , pageProps }: AppProps ) {
  const { store } = wrapper.useWrappedStore(pageProps);
  return (
    <Provider store={store}>
      <Layout>
        <Component {...pageProps} />
      </Layout>
    </Provider>
  )
}

export default CookBook
```

Figure 5.17: *Application file source with added store*

Now we can check if everything is correctly set up. To make it we can use the Chrome extension (or you can find the same for Firefox).You can install it at this link **https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en.** The extension should look like in *Figure 5.18:*

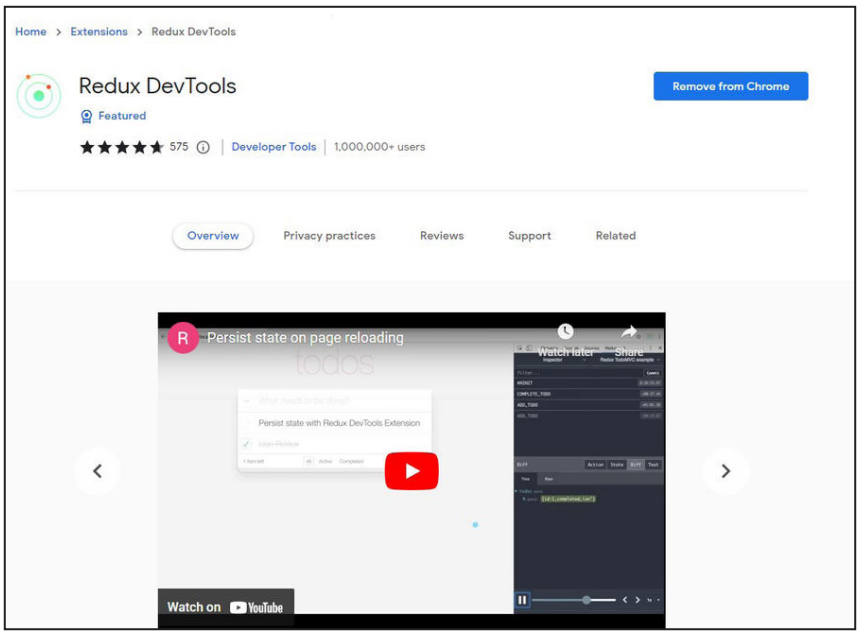


Figure 5.18: *Chrome extension to debug Redux in the browser*

Now after the page reboot we can open the developer tools and select the Redux extension. There will be information about the current store that we have like in *Figure 5.19:*

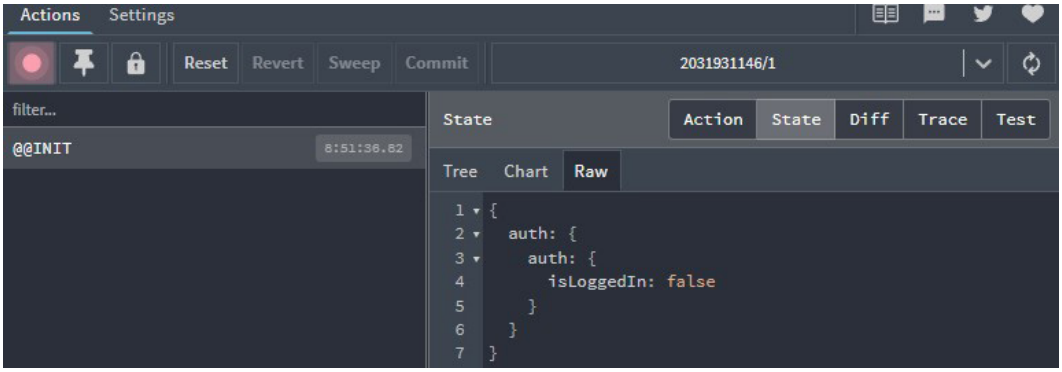


Figure 5.19: *Redux debug information for in developer tools*

Using the store for the authorization in our application

To do it we will do some changes in our code. First, we will rename the logged-in state in the strategy interface to fit the store object naming:

```
interface IUser {
    isLoggedIn: boolean;
    token: string;
    userProperties: Array<keyof typeof UserBuilderMethods>;
    userPropertiesActions?: string[]
}
```

Figure 5.20: *Change state name to isLoggedIn to fit the naming*

Next, we will need to update the store to collect more data for the user. To achieve it we will update **`Auth type`** in the store like this:

```
export type Auth = {
    isLoggedIn: boolean;
    token: string | null;
    userProperties: Array<string>
}
```

***Figure 5.21**: Update the store type to fit the requirements*

Do not forget to update the initial store as listed in *Figure 5.22*:

```
export const INITIAL_STATE: AuthState = {
    auth: {
        isLoggedIn: false,
        token: null,
        userProperties: []
    },
};
```

***Figure 5.22:** Initial store update*

To use the state of the login we need to do changes in the login service to make the state return from the method. Check *Figure 5.23* for the solution:

```
async login(user: string, password: string) {
        // Here we will provide the login logic depending on what strategy is selected
        const loginContext = new LoginContext(loginType);
        const loginState = await loginContext.useLogin(user, password)
        this.isLoggedIn = loginState.isLoggedIn;
        this.applicationUser = new ApplicationUser(loginState)
        loginState.userProperties.forEach((property: keyof typeof UserBuilderMethods) => {
            UserBuilderMethods[property] && this.applicationUser[UserBuilderMethods[property]]()
        })
        this.token = loginState.token;
        return loginState;
}
```

***Figure 5.23:** Updated login method in the service*

Now, we can open the login form file and update the **`loginAction`** function to get the state of the login after the method is triggered. As the login method is async we need to change the **`loginAction`** type to async listed here:

```
const loginAction = async (event: any) => {
        console.log('login');
        event.preventDefault();
        const loginState = await loginService.login(login, password);
        console.log('loginState', loginState)
}
```

Figure 5.24: *Form login method update*

After that, we will need to add a dispatch call to change the state. To do it open the login form file and add this as shown in *Figure 5.25:*

```
import { useAppDispatch } from '../../pages/hooks'
import { changeAuthState } from "../../pages/store/authSlice";
```

Figure 5.25: *Functions required to proceed*

Now, the **`loginAction`** function can dispatch the state and provide it in the action like this:

```
const dispatch = useAppDispatch();

const loginAction = async (event: FormEvent<HTMLFormElement>) => {
      console.log('login');
      event.preventDefault();
      const loginState = await loginService.login(login, password);
      dispatch(changeAuthState(loginState));
      console.log('loginState', loginState);
}
```

Figure 5.26: *Updated loginAction function*

Finally, when we will try to log in again in the login form we will see the changing state history in the developer tools extension as shown in *Figure 5.27:*

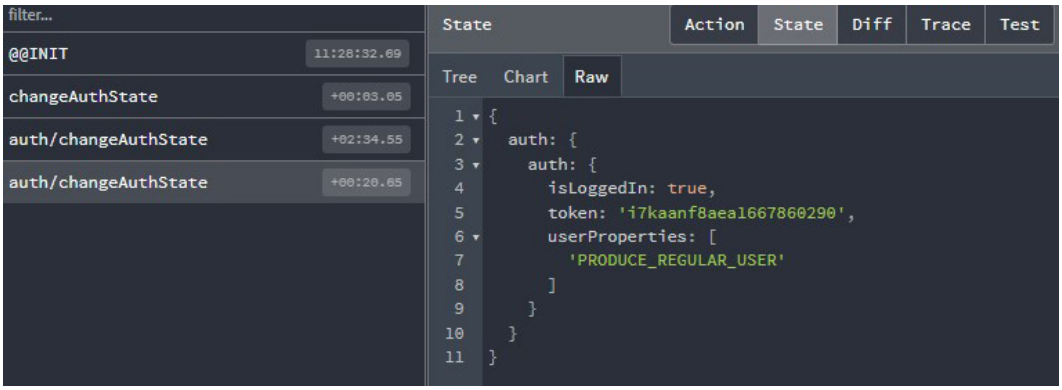


Figure 5.27: *State history in the Redux tools*

Connecting data API to state management

In our architecture, we do not have direct API calls because we use configurable strategies. But it is good news for us anyway. To connect API calls to the state management we will need to use middleware as the best practice solution.

Please follow *Figure 5.28* to update the store with middleware example:

```
const apiCallMiddleware = (store: RootState) =>
    (next: Dispatch<RootState>) =>
    (action: {type: string, payload : {save: boolean}}) => {
  console.log("action", {store, action});
  LoginService.getInstance().anyAPICall();
  next(action);
};

export const store = configureStore({
  middleware: (getDefaultMiddleware) =>
                    getDefaultMiddleware()
                      .concat(apiCallMiddleware),
  reducer: {
    [authSlice.name]: authSlice.reducer,
  },
  devTools: true,
});
```

Figure 5.28: *Update for the store/index.ts file*

After this update, each action call will be wrapped with a middleware function. We can filter actions and call different methods of service. As an example, we will add code from *Figure 5.29* to the login service.

```
anyAPICall() {
        console.log('here we can call an API');
}
```

Figure 5.29: *Dummy function to call it as an API call*

After the page reloads we can try to log in again and that is what we will see in the browser console:

```
00:08:55.636 trigger build                                    user-builder.ts?2efc:14
00:08:55.637 action ▶ {type: 'auth/changeAuthState', payload: {…}}   index.ts?f927:7
00:08:55.637 here we can call an API                          login.service.ts?d5fa:35
00:08:55.639 loginState                                       LoginForm.tsx?81f1:23
             ▶ {isLoggedIn: true, token: 'wrxx6tpvdod1667862535', userProperties: Array
               (1)}
```

Figure 5.30: *Text in the console that we added to the service method*

Now we can add as many API calls to the store as we want, depending on the action name. We can also provide the payload in the middleware function parameters so we could filter actions also by the payloads.

Conclusion

In this chapter, we learned how to connect the state management system to our NextJS application. We managed to use Test Driven Development and also figured out that for the API calls in the store we use the middleware instead a direct call from the reducer.

In the next chapter, we will use all the collected knowledge to create more internal pages for the application.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 6
Implementing internal pages using NextJS

Introduction

We did a great job in the previous chapters. But now it is time to step to the next level and create the system that will allow us to develop and publish internal pages. We will use all the knowledge that we get from before. For the exercise, we will not touch GraphQL possibilities in this chapter and use simple data fetch. We will do this because in the next chapters we will connect our application to the AWS Amplify environment and this part will be redundant.

Structure

- Creating the publishing system for the food blog. Basics
- Mocking list of articles and article description page
  - Creating mocks for internal pages
  - Splitting internal pages into components
- Creating the application structure and router for application pages
- Creating atoms and molecules

- Creating the TDD flow for all coding structures
  - Writing tests for page components
  - Writing tests for store
  - Writing tests for API
- Creating some API endpoints for the application
- Creating internal application pages
  - Creating an article list page
  - Creating an article item page
- Creating a CRUD system for articles
  - Separate public and private areas with NextJS
  - Redux store for data state and edit
  - Updating data in API
- Creating a multilingual tool for application in NextJS

Objectives

In this chapter, we will introduce how to start creating the publishing system from the very beginning. We will follow the guidelines, that we used before for mocking and test-driven development. Also, we will connect state management and API. And in the end, we will add the possibility to create a multilanguage application using NextJS.

Creating the publishing system for the food blog

Before we start our creative journey, let us agree on some requirements that will be used in the publishing system design:

- Each publication (we will call it an article) will be connected to one user.
- Each user can have an unlimited number of publications.
- Each publication will have a title, short description, text, and date of publication. (In the real-life application we will also have some images but to add this feature we will need an additional image server and so on. For our example, we will have only these text fields)

- The application will have an articles list page where all articles will be shown one by one unit, using the title, the description, and the publication date as content.
- By clicking on each article block user will be directed to the article page where he will see the article with the main text.
- To add the article to the navigation panel we will add the button to add the article.
- All new articles will be sorted by the publishing date without any other prioritization
- If we logged in as an article owner then we will have the possibility to edit or remove the article from the system. To add this functionality there should be special buttons for this in the list for each article and also in the exact article.

Mocking - List of articles and article description page

Now we have the requirement list we can start to create mocks for the pages. Let us assume that some functional elements will be visible only after authorization (like add, update, and delete buttons). We will separate each element using the atomic UI system as we did in previous chapters.

Until now, we do not store the state in the browser's local storage or any other storage. There are several different ways to solve this problem but it is out of the scope of this book.

To not reset the auth data we will simply store it in the local storage and use it on the application load. This way is not effective for a real-world production application. Please use any persist library for it or use the database of the browsers.

Creating mocks for internal pages

We will start from the article list page. On this page, we need to have several elements to the requirements. But as we have more than 1 page now we will add the navigation layout. Also do not forget that there will be an element that will be shown only after user authorization. Check *Figure 6.1* to see the mock for the article list that will be shown to the unauthorized user:

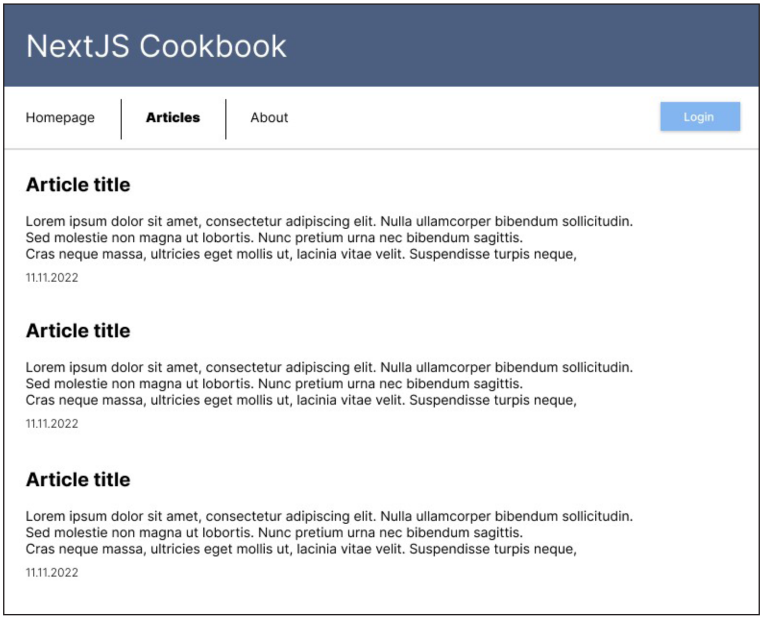


Figure 6.1: *Articles list page for the unauthorized user*

Please keep in mind that we do not have a logout function in our current system. Because of that, we will just show the add article button after login. Please check *Figure 6.2* to see the article list after the user login.

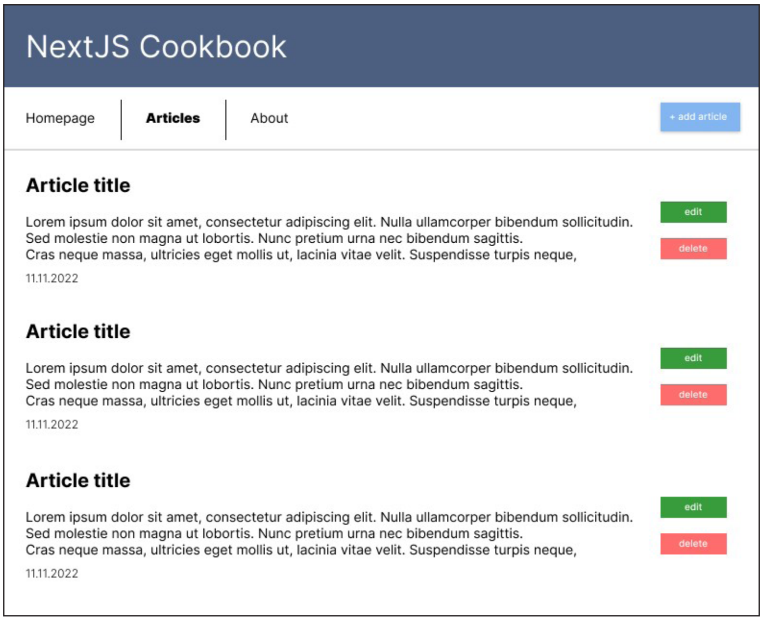


Figure 6.2: *Articles list page for the authorized user*

The whole article block will be clickable to enter the current article. That means we will not need any other navigation elements. As you can see we have '**`Edit`**' and '**`Delete`**' buttons in the list. The same buttons will be duplicated on the current article page. Please check *Figure 6.3* to see how the current article page will look for the unauthorized user.

Figure 6.3: *The article page for the unauthorized user*

The same for the authorized user can be observed in *Figure 6.4*:

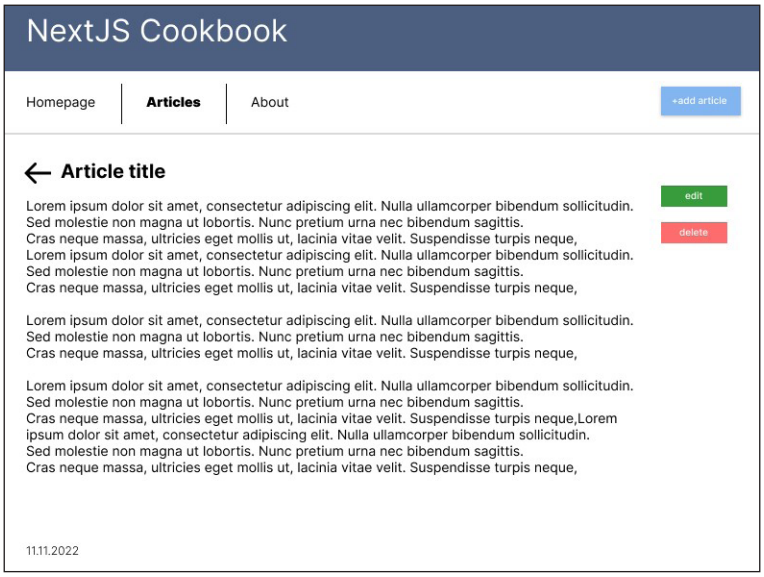


Figure 6.4: *The article page for the authorized user*

For the article edit, we will use a simple modal widow and form inside. You can see the mock in *Figure 6.5:*

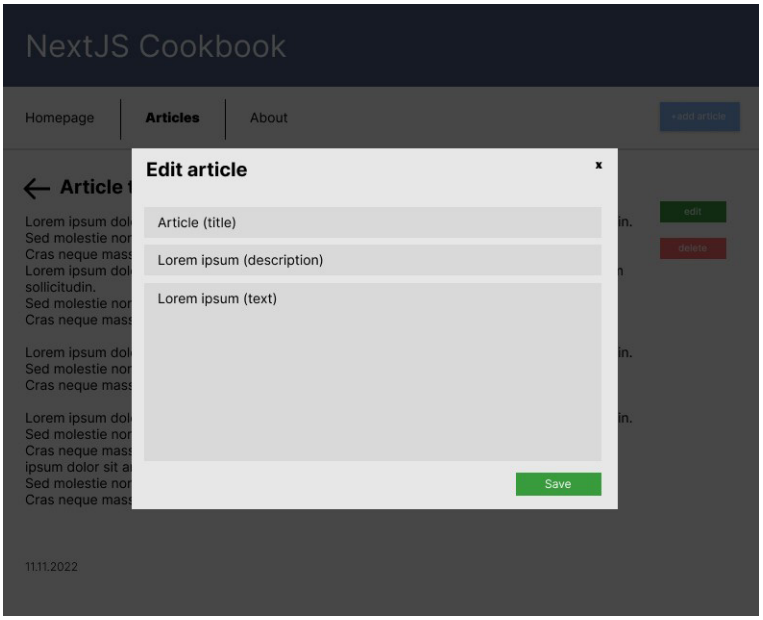


Figure 6.5: *Edit article modal window*

Splitting internal pages into components

To follow the atomic design pattern we need to separate everything into elements. Let us start with the articles list page to figure out what components we can create from it. On this page, we see a navigation bar that contains several atoms and one molecule there.

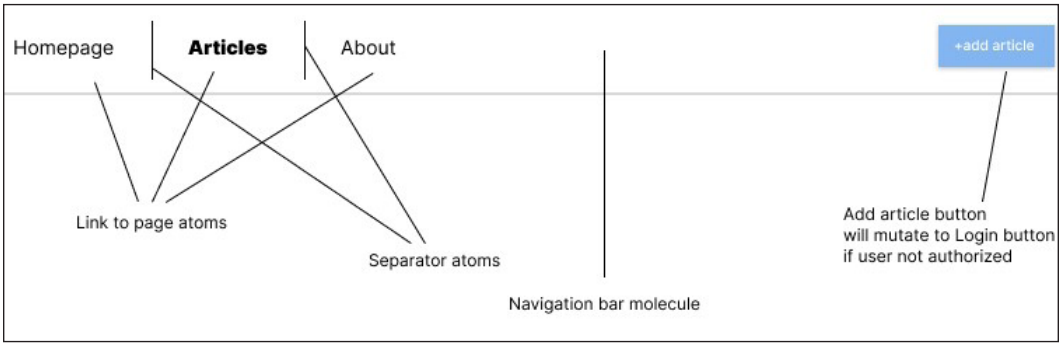


Figure 6.6: *Navigation bar separated using Atomic pattern.*

As you can see in *Figure 6.6* we will need to create the link to the page atom that will have visited state (bold font), the separator element between links. And also the add article button. This button will have an unauthorized state and lead to the login page. All these atoms will be wrapped by a navigation molecule that will have the bottom border and contain all the elements for the page.

The next part is the articles list and which can be observed in *Figure 6.7*:

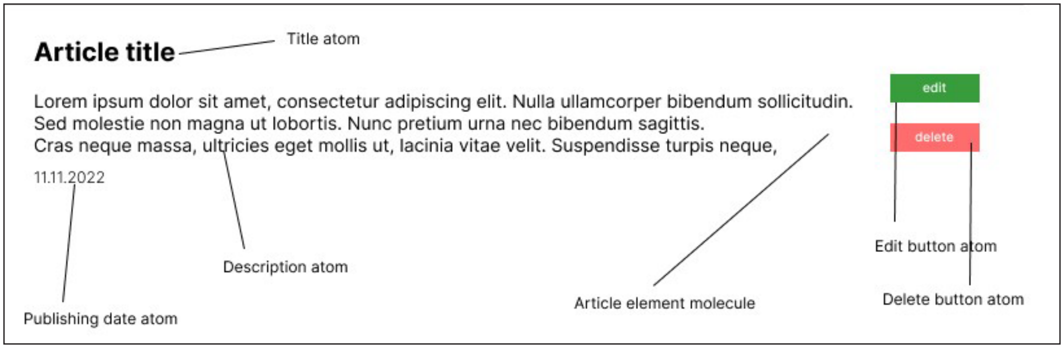


Figure 6.7: *Article element molecule*

As we can see we will need atoms for the title, description, and date. Also as the buttons for edit and delete. The atomic way of separating will help us to reuse atomic elements in other molecules. We can see it in *Figure 6.8* for the exact article page.

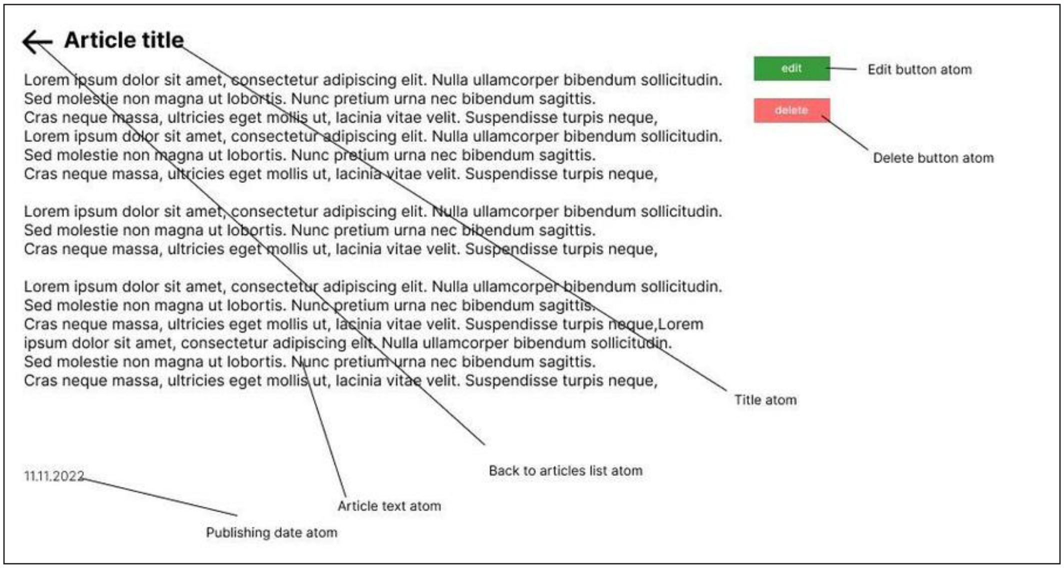


Figure 6.8: *Exact article molecule*

As you can see we can use the title, date, and buttons from the atomic system that we will create for the articles page. That will improve the maintainability of the system in the future and also reduce the number of code lines of the application.

Finally, we can point out several components for the edit modal that you can see in *Figure 6.9*.

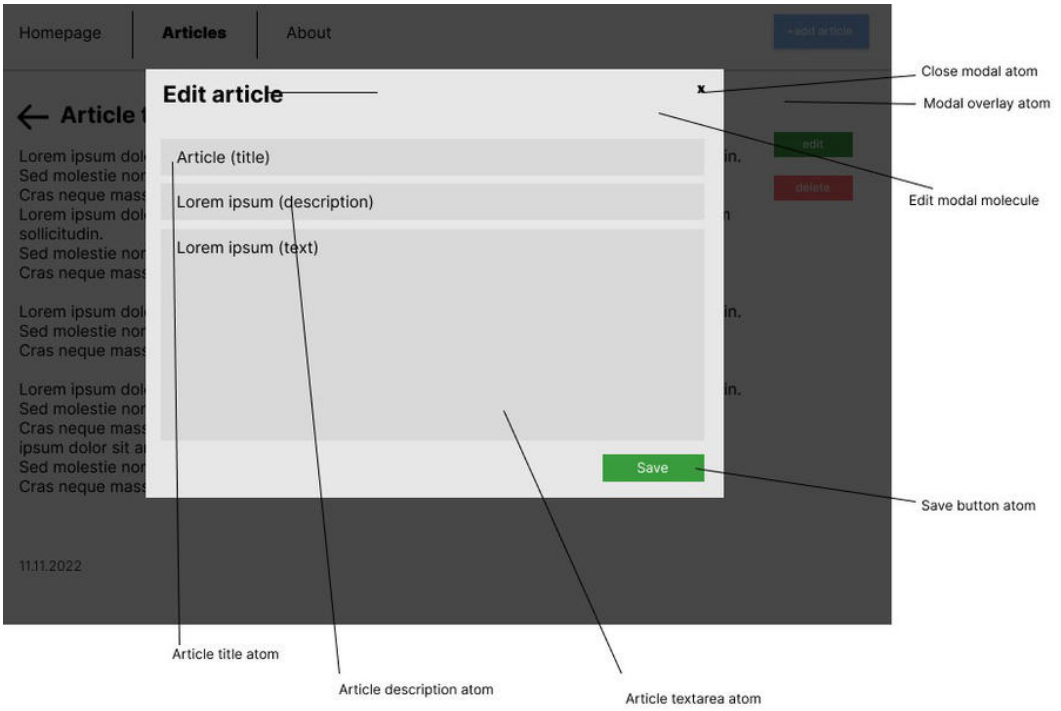


Figure 6.9: *Edit modal atoms*

We made a great job here and have a full list of required components that we could wrap with tests and make ready for production.

Creating the application structure for application pages

The next step in our process will be creating the files for the UI system. In *Figure 6.10* we can observe the result of previous research. We will create all required files in the UI folder.

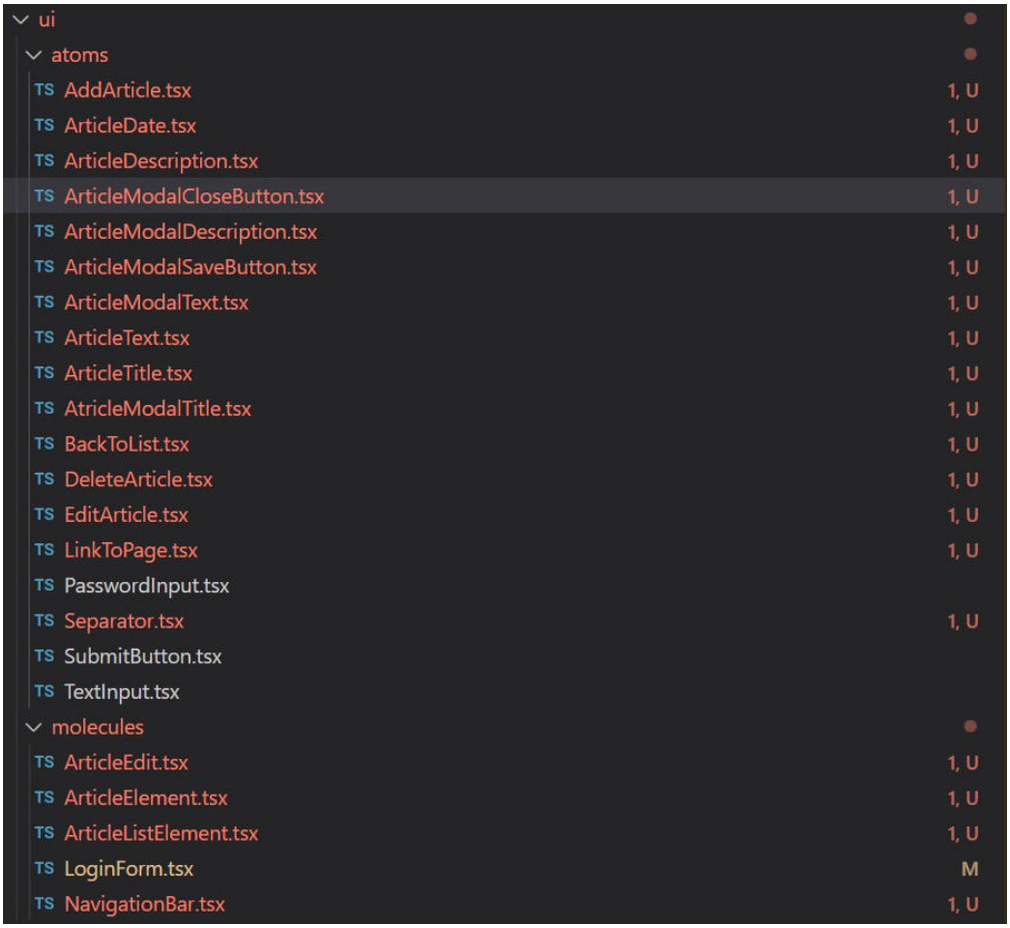


Figure 6.10: *UI folder file structure after the Atomic research.*

As in NextJS, the routing system is based on files we will need also to update the file structure to work with article pages like the following figure:

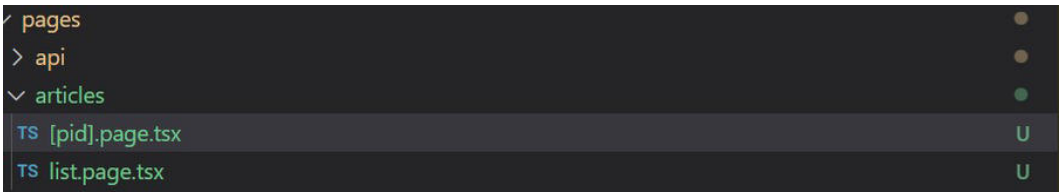


Figure 6.11: *Article pages structure update*

Before we start implementing the UI we need to add some features to the system. As you remember we will use a simple store for the auth state. To add it let us do some changes in several files.

Add a file with the name **`local-storage.ts`** in the pages/core folder. Put code from *Figure 6.12* into this file.

```
const updateStorage = (key: string, data: string) => {
    localStorage.setItem(key, data);
}

const getFromStorageByKey = (key: string, param?: string) => {
    if (!param) {
        return localStorage.getItem(key);
    }

    const storageToJSON = JSON.parse(localStorage.getItem(key) as string);
    return typeof storageToJSON === 'object' && storageToJSON[param];
}

export { updateStorage, getFromStorageByKey };
```

Figure 6.12: *Core file to work with local storage*

Next step we will need to call the **`updateStorage`** function when we will have the user data from the API. Make the login service function login same as presented in *Figure 6.13:*

```
async login(user: string, password: string) {
        const loginContext = new LoginContext(loginType);
        const loginState = await loginContext.useLogin(user, password)
        this.isLoggedIn = loginState.isLoggedIn;
        this.applicationUser = new ApplicationUser(loginState)
        loginState.userProperties.forEach((property: keyof typeof
UserBuilderMethods) => {
            UserBuilderMethods[property] &&
this.applicationUser[UserBuilderMethods[property]]()
        })
        this.token = loginState.token;
    /* Remark about LocalStorageKeys.LOGIN: do not forget to create the
enum with keys in core configuration and put value 'loginState' (or any
that you prefer) for the key LOGIN */
        updateStorage(LocalStorageKeys.LOGIN, JSON.stringify(loginState))
        return loginState;
    }
```

Figure 6.13: *Updated login service*

And now the main component file **`_app.page.tsx`** should be updated with the following code:

```
function CookBook({ Component , pageProps }: AppProps ) {
  const { store } = wrapper.useWrappedStore(pageProps);

  useEffect(() => {
    const authFromStorage = getFromStorageByKey(LocalStorageKeys.LOGIN)
    if (authFromStorage) {
      store.dispatch(changeAuthState(authFromStorage));
    }
  }, [store])

  return (
    <Provider store={store}>
      <Layout>
        <Component {...pageProps} />
      </Layout>
    </Provider>
  )
}
```

Figure 6.14: *Main component update*

Now each time we do reload the page the login state is not gone anyway and always exists in the application store as shown in *Figure 6.15:*

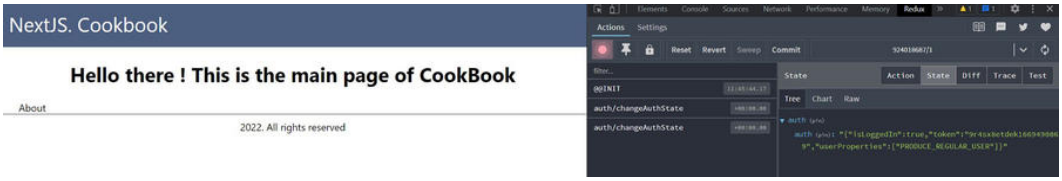


Figure 6.15: *Auth state exists if we manually route to the main page*

Creating atoms and molecules

We will start with more extensive parts and move to smaller parts so from molecules to atoms. We will have a lot of elements that look and act pretty much the same so we will put into text only more or less unique ones to have less identical code in the text (for example we have several buttons, and the only difference is the label and callback function). A good practice is to reduce the amount of the same code in your codebase so we will try to follow this guide as possible.

Atoms

We will use *Figure 6.10* as a list of required components that should be created. For now, we will leave everything as abstract as possible without any concrete solution to keep this example more focused on the process rather than on some amazing result.

For the styles please create the file with the name **`Atoms.module.scss`** in the styles folder.

The button for adding articles can be observed in *Figure 6.16*:

```
/* Please update your ui-types.d.ts with this interface
interface IAddArticleButton {
    openModal: () => void
}
*/

import { useEffect, useState } from "react";
import { useAppSelector } from "../../pages/hooks";
import { Labels, LocalStorageKeys } from "../../pages/core/configs"
import { selectAuthState } from "../../pages/store/authSlice";
import styles from '../../styles/Atoms.module.scss';

const AddArticleButton = ({ openModal }: IAddArticleButton) => {
    const [buttonLabel, setButtonState] = useState(Labels.SUBMIT);
    const isLoggedIn = useAppSelector(selectAuthState);

    useEffect(() => {
        if (isLoggedIn) {
            setButtonState(Labels.ADD_ARTICLE);
        }
        dispatch(changeArticleState(newArticle));
    }, [isLoggedIn]);
    return (
         <button
              className={styles.blueButton}
              type="button"
              onClick={ () => openModal() }
          >{!isLoggedIn ? Labels.SUBMIT : Labels.ADD_ARTICLE}</button>
    )
}

export default AddArticleButton;
```

Figure 6.16: *Add article button with mutable label*

Please also add the styles for this component as in *Figure 6.17*:

```
@import 'colors';
.blueButton {

    background-color: $button;
    padding: 1rem;
    border: none;
    border-bottom: 3px solid $gray;
    cursor: pointer;
    transition: 0.1s;
    margin: 0.2rem;
    position: relative;

    &:active {
        border-bottom: 1px solid $gray;
        margin-bottom: 0.3rem;
        bottom: -2px;
    }
}
```

Figure 6.17: *Styles for the button*

The dates will use this code for the display:

```
/* Please update your ui-types.d.ts with this interface
interface IDate {
    date?: string
};
*/

import styles from '../../styles/Atoms.module.scss';

const ArticleDate = ({ date }: IDate) => {
    return (
        <span className={styles.dates}>{date}</span>
    )
}

export default ArticleDate;
```

Figure 6.18: *Dates component*

```scss
.dates {
    color: $gray;
    font-size: 0.8em;
}
```

Figure 6.19: *Styles for the dates component*

The article title, description, and text will be combined with editing elements to have a group inside of the atom that is solving one task depending on the state that will contain a flag if we edit the element or not.

For the article title component use code from *Figure 6.20*. As you can see we separate the view for not authorized and authorized users. We will add a store for these components later.

```jsx
import { useState } from 'react';
import { useSelector } from 'react-redux';
import { selectAuthState } from '../../pages/store/authSlice';
import styles from '../../styles/Atoms.module.scss';

const ArticleTitle = ({ title, isEdit }: { title: string, isEdit: boolean})
=> {
    const [value, setValue] = useState(title)

    const onChangeHandler = (event: Partial<any>) => {
        const value = event.target.value;
        setValue(value);
    }
    return (
        <div className={styles.input}>
            {!isEdit &&
                <span className={styles.dates}>{title}</span>
            }
            {isEdit &&
                <input onChange={onChangeHandler} type="text" value=
{value}/>
            }
        </div>
    )
}

export default ArticleTitle;
```

Figure 6.20: *Article title component*

Add this style to the styles file.

```
.input {
    span {
        color: $gray;
        font-size: 1em;
    }
    input {
        color: $gray;
        font-size: 1em;
        padding: 0.5rem;
        width: 100%;
    }
}
```

Figure 6.21: *Styles for the input element*

The article description is the same as the title in our example, so you can just copy and paste the whole code. Do not forget to rename the component and properties like the following figure:

```
const ArticleDescription = ({ description, isEdit }: { description: string,
isEdit: boolean }) => {
    /*
        same code here
    */
}

export default ArticleDescription;
```

Figure 6.22: *Article description component*

Article text is using **textarea** as an input like the following figure:

```
import { useState } from 'react';
import { useSelector } from 'react-redux';
import { selectAuthState } from '../../pages/store/authSlice';
import styles from '../../styles/Atoms.module.scss';

const ArticleText = ({ text, isEdit }: { text: string, isEdit: boolean}) =>
{
    const [value, setValue] = useState(text)

    const onChangeHandler = (event: Partial<any>) => {
        const value = event.target.value;
        setValue(value);
    }
    return (
        <div className={styles.input}>
            {!isEdit &&
                <span className={styles.dates}>{text}</span>
            }
            {isEdit &&
                <textarea onChange={onChangeHandler} value={value}>
</textarea>
            }
        </div>
    )
}

export default ArticleText;
```

Figure 6.23: *Article text component*

Do not forget about styles and add this style input to your input class like the following figure:

```
.input {
    /*
        ...
    */
    textarea {
        color: $gray;
        width: 100%;
        font-size: 1em;
        padding: 0.5rem;
        width: 100%;
        height: 5rem;
    }
}
```

Figure 6.24: *Textarea styles to insert into .input class*

Let us move forward and create the close modal button like the following figure:

```
import styles from '../../styles/Atoms.module.scss';

const ArticleModalCloseButton = ({ closeModal }: { closeModal: () => void }) => {
    return (
        <button className={styles.close} type="button" onClick={ () => closeModal() }>X</button>
    )
}

export default ArticleModalCloseButton;
```

Figure 6.25: *Close button component*

Update the styles file with the class to add some styles to the button as shown in *Figure 6.26*:

```
.close {
    line-height: 1;
    background: no-repeat;
    border: none;
    font-size: 1.2em;
    cursor: pointer;
    transition: 0.1s;

    &:active {
        font-weight: bold;
    }
}
```

Figure 6.26: *Close button class in styles file*

The back-to-list button will look like this:

```
import styles from '../../styles/Atoms.module.scss';

const BackToListButton = ({ backToList }: { backToList: () => void }) => {
    return (
        <button className={styles.back} type="button" onClick={backToList}>←</button>
    )
}

export default BackToListButton;
```

Figure 6.27: *Back to list button*

The styles for this button shown in *Figure 6.28:*

```
.back {
    background: none;
    border: none;
    font-size: 2em;
    transition: 0.5s;
    cursor: pointer;

    &:active {
        font-size: 3em;
    }
}
```

Figure 6.28: *Style class for the back-to-list button*

The delete article button will look like this:

```
/* Please update your ui-types.d.ts with this interface
interface IArticleActions {
    saveArticle?: () => void
    deleteArticle?: () => void
    editArticle?: () => void
};
*/

import { Labels, LocalStorageKeys } from "../../pages/core/configs"
import styles from '../../styles/Atoms.module.scss';

const DeleteArticleButton = ({ deleteArticle }: IArticleActions) => {
    return (
        <button
            className={styles.deleteButton}
            type="button"
            onClick={ () => deleteArticle?.() }
        >{Labels.DELETE}</button>
    )
}

export default DeleteArticleButton;
```

Figure 6.29: *Edit article button*

Styles for this button will be also short like this:

```
.deleteButton {
    @extend .blueButton;
    color: $white;
    background-color: $red;
    padding: 0.5rem 1rem;
}
```

Figure 6.30: *Style class for delete article button*

The edit button will be the same but with a different name of components and style as shown in *Figure 6.31*:

```
const EditArticleButton = ({ editArticle }: IArticleActions) => {
    /*
        same code
    */
}

export default EditArticleButton;
```

Figure 6.31*: Edit article button component*

For style, we will extend the delete button but change the color. Now both buttons are the same with only one difference as shown in *Figure 6.32:*

```
.editButton {
    @extend .deleteButton;
    background-color: $green;
}
```

Figure 6.32: *Style for edit article class*

We also have a separator between links in navigation:

```
import styles from '../../styles/Atoms.module.scss';
const Separator = () => {
    return (
        <div className={styles.separator}></div>
    )
}
export default Separator;
```

Figure 6.33: *Separator component*

The styles class for this component will look like this:

```
.separator {
    border-right: 1px solid $gray;
    width: 1px;
    height: 2.5rem;
}
```

Figure 6.34: *Separator component style class*

The last component in the list will be the navigation link and the code is in *Figure 6.35*:

```
import Link from "next/link";
import styles from '../../styles/Atoms.module.scss';

const LintToPage = ({ title, link }: {title : string, link: string}) => {
    return (
        <Link
            className={styles.link}
            href={link}>
                {title}
        </Link>
    )
}

export default LintToPage;
```

Figure 6.35: *Link to the page component*

Styles for this component are in *Figure 6.36:*

```
.link {
    font-size: 1em;
    color: $gray;
}
```

Figure 6.36: *Style class for the link to the page component*

Molecules

The navigation bar will contain the code from *Figure 6.37:*

```
/* Please update your ui-types.d.ts with this interface
interface INavigationParams {
    navigation : Array<INavigation>
};
*/

import styles from '../../styles/Atoms.module.scss';
import LinkToPage from '../atoms/LinkToPage';
import Separator from '../atoms/Separator';

const NavigationBar = ({ navigation } : INavigationParams) => {
    return (
        <nav className={styles.nav}>
              {
                navigation?.map((navElement: INavigation) => {
                  return (
                    <div>
                        <LinkToPage
                            title={navElement.title}
                            link={navElement.link}
                        />
                        <Separator />
                    </div>
                  )
              }
        </nav>
    )
}

export default NavigationBar;
```

Figure 6.37: *Navigation bar component*

Styles for this component are listed in *Figure 6.38:*

```
.nav {
    display: flex;
    align-items: center;
    div {
        display: flex;
        align-items: center;
        padding: 1rem;
    }
    div:last-child .separator {
        display: none;
    }
}
```

Figure 6.38: *Style class from the navigation*

For the article item in the list use the code listed in *Figure 6.39*:

```
import Link from 'next/link';
import styles from '../../styles/Atoms.module.scss';
import ArticleDate from '../atoms/ArticleDate';
import ArticleDescription from '../atoms/ArticleDescription';
import ArticleTitle from '../atoms/ArticleTitle';
import DeleteArticleButton from '../atoms/DeleteArticle';
import EditArticleButton from '../atoms/EditArticle';

const ArticleListElement = ({ article, isLoggedIn } : {article : IArticle, isLoggedIn: boolean }) =>
{   return (
       <section className={styles.articleListElement}>
        <Link href={`/articles/${article.id}`} className={styles.linkToDiv}>
            <div><ArticleTitle isEdit={false} title={article.title} /></div>
            <div><ArticleDescription isEdit={false} description={article.description} /></div>
            <div><ArticleDate date={article.publishingDate}/></div>
        </Link>

        {isLoggedIn &&
            <div>
                <div>
                    <EditArticleButton editArticle={() => {}} /> <br />
                </div>
                <div>
                    <DeleteArticleButton deleteArticle={() => {}} /> <br />
                </div>
            </div>
        }
       </section>
    )
}

export default ArticleListElement;
```

Figure 6.39: *Article list item component code*

Add these styles to the atomic styles file:

```
.articleListElement {
    display: flex;
    align-items: center;
    border: 1px solid;
    justify-content: space-between;
    padding: 0.5rem;
}

.linkToDiv {
    display: block;
    width: 100%;
}
```

Figure 6.40: *Styles for the article list item component*

Now we need to do updates in the **`Edit`** button component and add a modal there. Please collect the updated code in *Figure 6.41:*

```
import { useState } from "react";
import { Labels, LocalStorageKeys } from "../../pages/core/configs"
import styles from '../../styles/Atoms.module.scss';
import ArticleEdit from "../molecules/ArticleEdit";
import ArticleDescription from "./ArticleDescription";
import ArticleModalCloseButton from "./ArticleModalCloseButton";
import ArticleModalSaveButton from "./ArticleModalSaveButton";
import ArticleText from "./ArticleText";
import ArticleTitle from "./ArticleTitle";

const EditArticleButton = ({ editArticle, article }: { editArticle: ()=> void, article: IArticle}) => {
    const [showModal, setModalState] = useState(false);
    return (
        <>
        <button
            className={styles.editButton}
            type="button"
            onClick={ () => setModalState(true) }
        >
            {Labels.EDIT}
        </button>
        {showModal &&
            <div id="edit" className={styles.modal}>
                <div className={styles.modalContent}>
                    <div className={styles.modalContent__first}>
                       <ArticleEdit isEdit={true} article={article} editArticle={editArticle} />
                    </div>
                    <div>
                        <ArticleModalCloseButton closeModal={ () => setModalState(false) } />
                    </div>
                </div>
            </div>
        }
        </>
    )
}

export default EditArticleButton;
```

Figure 6.41: *Updated code for the Edit button*

Please also add the following code to the styles:

```
.modal {
    display: block;
    position: fixed;
    z-index: 1;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    overflow: auto;
    background-color: $black;
    background-color: rgba(0,0,0,0.4);
}

.modalContent {
    background-color: $white;
    margin: 15% auto;
    padding: 20px;
    border: 1px solid $gray;
    width: 80%;
    display: flex;
    justify-content: space-between;
    &__first {
        width: 100%;
        div {
            margin-bottom: 1rem;
        }
    }
}
```

Figure 6.42: *The styles classes for the modal window*

The last molecule is the **`Article Edit`** component. Code can be collected in *Figure 6.43:*

```
/* Please update your ui-types.d.ts with this interface
interface IArticelEdit {
    article: IArticle
    isEdit: boolean
    editArticle: any
};
*/

import ArticleDescription from '../atoms/ArticleDescription';
import ArticleModalSaveButton from '../atoms/ArticleModalSaveButton';
import ArticleText from '../atoms/ArticleText';
import ArticleTitle from '../atoms/ArticleTitle';

const ArticleEdit = ({ article, isEdit, editArticle }: IArticelEdit) => {

    return (
        <>
         <ArticleTitle title={article.title} isEdit={true} />
         <ArticleDescription description={article.description} isEdit={true} />
         <ArticleText text={article.text} isEdit={true} />
         <ArticleModalSaveButton saveArticle={editArticle} />
        </>
    )
}

export default ArticleEdit;
```

Figure 6.43: *Article edit component*

Creating the TDD flow for all coding structures

The guidelines from the previous chapters are leading us to create tests first. We will plan to cover article pages with short tests before we start creating the page then cover the store and in the end, cover the API endpoints (for the last one we will do some smoke tests as we do not need the full possibilities of the API in this chapter because it's out of scope).

Writing tests for page components

Because we do not have many elements on the page that appear without data we will need to test only that component is rendered well as we did it before. (*Figure 6.44)*

```
import React from 'react'
import { screen } from '@testing-library/react'
import { renderWithProviders } from '../utils'
import ListPage from '../pages/articles/list.page'

describe('List Articles', () => {
  it('renders a list page', () => {
    renderWithProviders(<ListPage />)

    const heading = screen.getByRole('heading', {
      name: /Articles list/i,
    });

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

Figure 6.44: *Render test for list articles page*

Please add the code provided in *Figure 6.45* to add a test for the exact article page:

```
describe('Articles', () => {
  /*
  */

  it('renders an article page', () => {
    renderWithProviders(<ArticlePage />)

    const heading = screen.getByRole('button', {name: 'back to
list'});
    expect(heading).toBeInTheDocument();
  })

})
```

Figure 6.45: *Exact article page test*

Writing tests for store

By design, we do not have any dynamic components on the pages. To have an example, let us add the test for the selected article here. We will add it in the future.

```
import articleSlice, {
    changeArticleState,
    INITIAL_STATE,
    ArticleState
} from '../articleSlice';

describe("Article Slice", () => {
  describe("Article check function", () => {
    it("should change artile in store", () => {
        const state: ArticleState = {
          isNew: true
        };
          const action = changeArticleState(state);
          const expectedResult: ArticleState = state;
          const actualResult = articleSlice.reducer(INITIAL_STATE, action);
          expect(actualResult).toEqual(expectedResult);
    });
  });
});
```

Figure 6.46: *Article`s state test*

Leave it for now. We will come back to it in the future.

Writing tests for API

Please note that we will not create any functions for the API calls and in this section will call the data directly from the mock using the API possibilities of NextJS. That means that in this block we will skip the API test but come back to it in the E2E section of the book.

Creating some API endpoints for the application

To add the API endpoints we will need to add some folder structures as you can see in *Figure 6.47:*

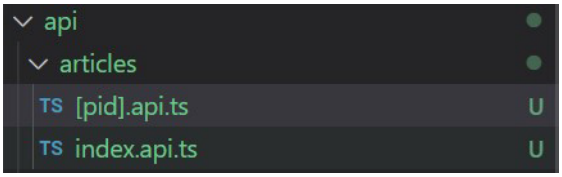


Figure 6.47: *Articles file structure*

For the article list we will use an index file with this code:

```
import type { NextApiRequest, NextApiResponse } from
'next'
import { getMock } from "../../mocks";

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<any>
) {
 res.status(200).json(getMock.new_articles);
}
```

Figure 6.48: *Articles list endpoint*

For the exact article, we will use **`[pid].api.ts`** file with the following code inside:

```
import type { NextApiRequest, NextApiResponse } from
'next'
import { getMock } from "../../mocks";

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<any>
) {
 const { pid } = req.query
 res.status(200).json(getMock.new_articles[Number(pid)]);
}
```

Figure 6.49: *Exact article endpoint file*

Creating internal application pages

We are ready to create the pages as we have made all possible preparations before we do a soft start. We have all the required tests and all required API endpoints.

Creating an article list page

To add the codebase to this page we need to do the update for the general view and add the navigation bar to it. Update the navigation bar component using the code from *Figure 6.50:*

```
import { useRouter } from 'next/router';
import styles from '../../styles/Atoms.module.scss';
import AddArticleButton from '../atoms/AddArticle';
import LinkToPage from '../atoms/LinkToPage';
import Separator from '../atoms/Separator';


const NavigationBar = ({ navigation } : {navigation : Array<INavigation>}) => {
    const router = useRouter();
    return (
        <nav className={styles.nav}>
            <div>
                {navigation && navigation.map((navElement: INavigation) => {
                    return ( <div key={navElement.link} className={`${router.asPath === navElement.link ?
styles.active : ""}`}>
                                    <LinkToPage
                                        title={navElement.title}
                                        link={navElement.link}
                                    />
                                    <Separator />
                                </div> )
                })}
            </div>
            <div>
                <AddArticleButton openModal={ () => {} } />
            </div>
        </nav>
    )
}

export default NavigationBar;
```

Figure 6.50: *Updated navigation barcode*

Then please update the **`layout.tsx`** file with code that you will find in *Figure 6.51:*

```
import styles from '../../styles/layout.module.scss'
import NavigationBar from '../../ui/molecules/NavigationBar'

export default function Layout({ children } : Partial<any>) {
    const navigation = [
      {title: 'Home', link: '/'},
      {title: 'Articles', link: '/articles'},
      {title: 'About', link: '/about'}
    ]
    return (
      <>
        <header className={styles.header}>NextJS. Cookbook</header>
        <nav>
          <NavigationBar navigation={navigation} />
        </nav>
        <main>{children}</main>
        <footer className={styles.footer}>2022. All rights
reserved.</footer>
    )
  }
```

Figure 6.51: *Updated layout file*

Now we are ready to implement the list page using the code from *Figure 6.52*:

```
/* Please update your ui-types.d.ts with this interface
interface IListPage {
    notFound?: boolean
    data?: IArticle[]
};

*/

import type { NextPage } from 'next'
import ArticleListElement from '../../ui/molecules/ArticleListElement';
import { Fragment } from 'react';
import { selectAuthState } from '../store/authSlice';
import { useAppSelector } from '../hooks';

const ListPage: NextPage = ({ data, notFound }: IListPage) => {
  const isLoggedIn = useAppSelector(selectAuthState);

  return (
    <section>
      <h1>Articles list</h1>
      {
        data?.map((item: any) => {
          return <Fragment key={item.id}>
                  <ArticleListElement
                    isLoggedIn={isLoggedIn}
                    article={item}
                    />
                  </Fragment>
        })
      }
      {
        notFound && <span>No articles found</span>
      }
    </section>
  )
}

export default ListPage
```

Figure 6.52: *Component code without data fetching*

To fetch the data add the following code to your component:

```
/*
    component code from previuos Figure
*/

export async function getServerSideProps() {
  try {
    // This part can be changed to service
    const { data } = await axios.get<any>(
      'http://localhost:3005/api/articles'
    );
    if (!data) {
      return {
        props: { notFound: true },
      };
    }

    return {
      props: { data },
    };
  } catch (error) {
    return {
      props: { notFound: true },
    };
  }
};

export default ListPage
```

Figure 6.53: *Get server-side props function for the component*

Create an article item page

Now we can create the article page using the same pattern that we used in the list page. Please collect the full code of the internal article page from *Figure 6.54:*

```
import axios from 'axios'
import type { NextPage } from 'next'
import { useSelector } from 'react-redux'
import ArticleDate from '../../ui/atoms/ArticleDate'
import ArticleText from '../../ui/atoms/ArticleText'
import ArticleTitle from '../../ui/atoms/ArticleTitle'
import DeleteArticleButton from '../../ui/atoms/DeleteArticle'
import EditArticleButton from '../../ui/atoms/EditArticle'
import { selectAuthState } from '../store/authSlice'
import styles from '../../styles/Articles.module.scss';
import BackToListButton from '../../ui/atoms/BackToList'
import { useRouter } from 'next/router'

const ArticlePage: NextPage = ({ data, notFound }: any) => {
  const isLoggedIn = useSelector(selectAuthState);
  const router = useRouter();
  const routeBack = () => {
    router.push('/articles');
  }
  return (
    <section className={styles.section}>
      <div>
        <div className={styles.title}>
          <BackToListButton backToList={ () => routeBack()} />
          <h1>
            <ArticleTitle title={data.title} isEdit={false} />
          </h1>
        </div>
        <p>
        <ArticleText text={data.text} isEdit={false} />
        </p>
        <div>
          <ArticleDate date={data.publishingDate} />
        </div>
      </div>
      {isLoggedIn && <div>
                <div>
                    <EditArticleButton article={data} editArticle={() => {}} /> <br
/>              </div>
                <div>
                    <DeleteArticleButton deleteArticle={() => {}} /> <br />
                </div>
      </div> }
    </section>
  )
}

export async function getServerSideProps(context) {
  const { pid } = context.query;
  console.log(context);
  try {
    // This part can be changed to service
    const { data } = await axios.get<any>(
      'http://localhost:3005/api/articles/' + pid
    );
    if (!data) {
      return {
        props: { notFound: true },
      };
    }

    return {
      props: { data },
    };
  } catch (error) {
    return {
      props: { notFound: true },
    };
  }
};


export default ArticlePage
```

Figure 6.54: *Exact article page component code*

Please also create the **`Article.module.scss`** file with these styles.(*Figure 6.55*)

```scss
.section {
    display: flex;
    justify-content: space-between;
}

.title {
    display: flex;
}
```

Figure 6.55: *Styles for the exact article*

Now when you visit the articles page you should see the following figure:

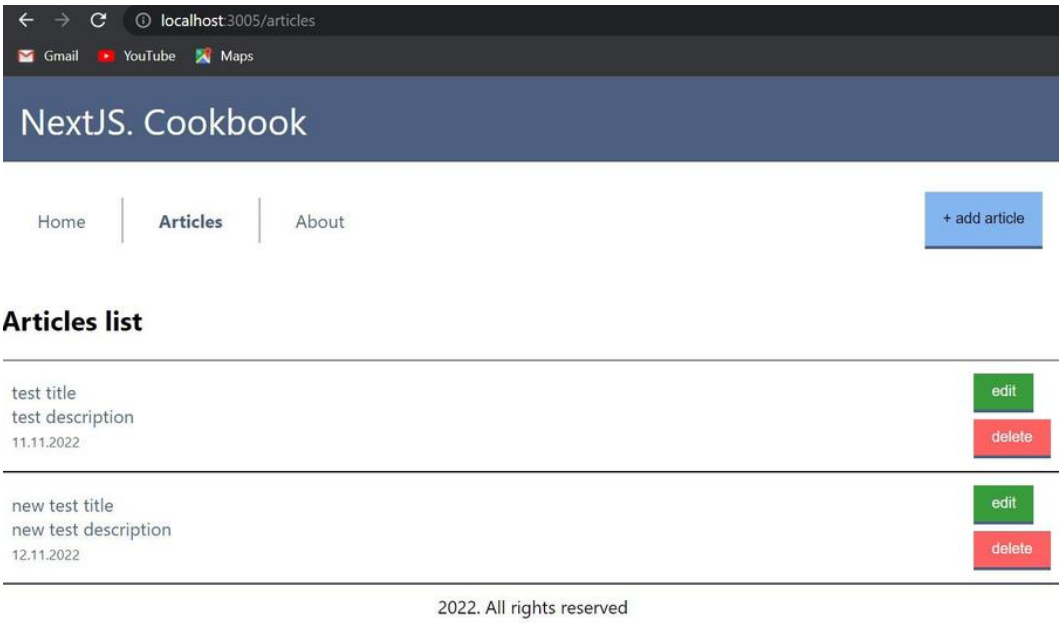


Figure 6.56: *Articles list page*

After all manipulations if you make click on any article title you should be followed to the page that will look like in *Figure 6.57*:

Figure 6.57: *Exact article page*

If we will not be logged in it will hide the action buttons and show the login button in the navigation.

Creating a CRUD system for articles

It is time to bring some life to our application. We can see the list of articles and also can enter each one. But we also need to have the possibility to have a UI for the create and update actions.

Separate public and private areas with NextJS

To check if we have a strong separation let us remove everything from the localStorage of the browser and reload the page. After that please visit the articles page (should be something like **http://localhost:3005/articles** this link). Now to have a short problem - the Login button is not active so we cannot get into the private area. To fix it we need to add a small update to this button:

```
return (
        <>
            {isLoggedIn &&
                <button
                    className={styles.blueButton}
                    type="button"
                    onClick={ () => openModal() }
                >{Labels.ADD_ARTICLE}</button>
            }
            {!isLoggedIn && <Link className={styles.blueButton}  href={"/login"}>{Labels.SUBMIT}</Link>}
        </>
    )
```

Figure 6.58: *Update for the AddArticle component*

Update the code in the **`AddArticle.tsx`** file using provided code from *Figure 6.58.* After that, if you click this button you will be directed to the login form page.

Let us also structure the styles a little to make the login page look like the other pages. Please add these styles to **`LoginForm.module.scss:`**

```
.LoginForm {
    max-width: 50%;
    margin: 0 auto;
    input {
        width: 100%;
        padding: 0.5rem;
        margin-bottom: 1rem;
    }
    button {
        padding: 1rem;
    }
}
```

Figure 6.59: *Update for the login page styles*

Please also update the login page render part as provided in *Figure 6.60:*

```
return (
    <article className="content">
      <h1>Login</h1>
      <LoginForm />
    </article>
  )
```

Figure 6.60: *Updated render part for the login page*

After that you will see that the page looks a little bit better than before:

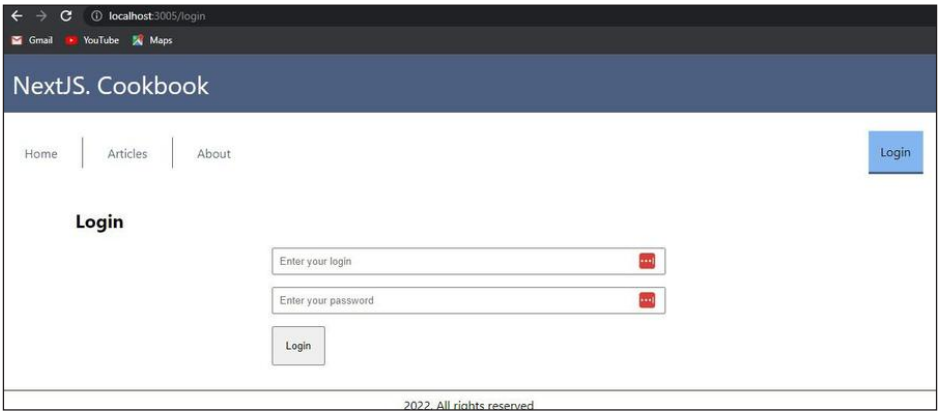


Figure 6.61: *Small improvement for the login page*

But we still have the lack of the process here as after the login nothing is happening. To fix it let us route to the articles page after we successfully logged in. To achieve it we will update the **`LoginForm`** component with **`route.push`** after the successful login like in *Figure 6.62:*

```
const LoginForm = () => {
    // same code as in file
    const router = useRouter();

    // same code as in file
    const loginAction = async (event: any) => {
        // same code as in file
        if (loginState.isLoggedIn) {
            router.push('/articles');
        }
       // same code as in file
    }
```

Figure 6.62: *Update for the login form component*

Now after we do successfully login into the system we will be redirected to the articles page.

Redux store for data state and edit

As you remember we are having the state test for the article that still not exists. Let us change the situation and create the slice for it like in *Figure 6.63:*

```
import { createSlice, PayloadAction } from "@reduxjs/toolkit";
import { RootState } from ".";

export type ArticleState = {
    id: number
    title: string
    description: string
    text: string
    publishingDate: string
    isNew: boolean
}

export const INITIAL_STATE: ArticleState = {
    id: 0,
    title: '',
    description: '',
    text: '',
    publishingDate: '',
    isNew: false
};

const articleSlice = createSlice({
  name: "article",
  reducers: {
    changeArticleState: (state: RootState, action: PayloadAction<ArticleState>) => {
        const newArticleState = action.payload;
        state = {...newArticleState };
        return state;
    }
  },
  initialState: INITIAL_STATE,
});

export const selectArticleState = (state: RootState) => {
    return state.article;
}
export const { changeArticleState } = articleSlice.actions;
export default articleSlice;
```

Figure 6.63: *Article state slice*

Now in the VS Code if we open the article slice test file we will see that all tests are passed and have a green indicator like in *Figure 6.64*:

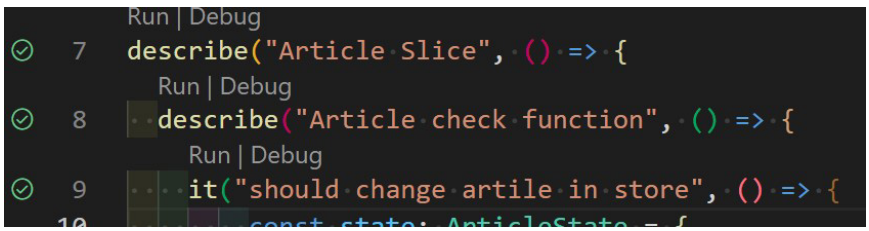


Figure 6.64: *Tests for the article slice*

Now we can add some functionality to the add article button as we still do not have a modal for it. Use this code to update the add article button to have the modal in it:

```
import Link from "next/link";
import { useEffect, useState } from "react";
import { Labels } from "../../pages/core/configs"
import { useAppDispatch, useAppSelector } from "../../pages/hooks";
import { changeArticleState } from "../../pages/store/articleSlice";
import { selectAuthState } from "../../pages/store/authSlice";
import styles from '../../styles/Atoms.module.scss';
import ArticleEdit from "../molecules/ArticleEdit";
import ArticleModalCloseButton from "./ArticleModalCloseButton";

const AddArticleButton = ({ openModal }: IAddArticleButton) => {
    const [buttonLabel, setButtonState] = useState(Labels.SUBMIT);
    const [showModal, setModalState] = useState(false);
    const isLoggedIn = useAppSelector(selectAuthState);
    const dispatch = useAppDispatch();
    const newArticle = {
        id: -1,
        title: '',
        description: '',
        text: '',
        publishingDate: '',
        isNew: true
    };

    const saveData = (data: IArticle) => {
        console.log('data to save', data)
    }

    useEffect(() => {
        if (isLoggedIn) {
            setButtonState(Labels.ADD_ARTICLE);
        }
        dispatch(changeArticleState(newArticle));
    }, [newArticle]);
    return (
        <>
            {isLoggedIn &&

                    <button
                        className={styles.blueButton}
                        type="button"
                        onClick={ () => setModalState(true) }
                    >{Labels.ADD_ARTICLE}</button>
            }

            {showModal &&
                    <div id="edit" className={styles.modal}>
                        <div className={styles.modalContent}>
                            <div className={styles.modalContent__first}>
                                <ArticleEdit isEdit={true} article={newArticle} editArticle={saveData} />
                            </div>
                            <div>
                                <ArticleModalCloseButton closeModal={ () => setModalState(false) } />
                            </div>
                        </div>
                    </div>

            }
            {!isLoggedIn && <Link className={styles.blueButton}  href={"/login"}>{Labels.SUBMIT}</Link>}
        </>
    )
}

export default AddArticleButton;
```

Figure 6.65: *Updated article add button component*

Updating data in API

We have enough interactivity for the internal pages so we can try to send some data into the API. We will also do it directly in this lesson. To create the payload for the action we need to store it somewhere. As you remember we store the current article data in the store. We can use it for sending data into an API to save the data.

Please update your inputs with the code provided in *Figure 6.66:*

```
const ArticleTitle = ({ title, isEdit }: { title: string, isEdit: boolean}) =>
{   // same code
    const currentArticle = useSelector(selectArticleState);
    const dispatch = useAppDispatch();

    const onChangeHandler = (event: Partial<any>) => {
        // same code
        dispatch(changeArticleState({...currentArticle, title: value}));
    }
    // same code
}

export default ArticleTitle;
```

Figure 6.66: *The code for input update*

Use the same code for description and text components with only property changes depending on the component name.

Now we need to add action to the save button. As you remember we are using middleware in this example so any change in the store will be wrapped with an API call. For the save button we just need to create the store update event and moderate the action and payload.

```
const EditArticleButton = ({ editArticle, article }: { editArticle: ()=> void, article: ArticleState}) => {
    // same code

    const saveArticle = async () => {
        dispatch(changeArticleState({...article, save: true}));
    }

    return (
        // same code
              <ArticleEdit isEdit={true} article={article} editArticle={saveArticle} />
        //same code
    )
}
```

Figure 6.67: *Update for the edit article button*

In the **EditArticleButton** component we need to make an update and put the function of store change into the **ArticleEdit** component like in *Figure 6.67*. Now each time we press the Save button store will be updated. We need to operate this event in the index file like this:

```
const apiCallMiddleware = (store: any) => (next: any) => (action: any) => {
  // same code
  let data = null;
  if (action.type === 'article/changeArticleState' && action.payload.save)
{   data = action.payload
  }
  LoginService.getInstance().anyAPICall(data);
  // same code
};
```

Figure 6.68: *Update of the API call event*

Now when we press the save button we will see that API was called with payload data inside like this:

Figure 6.69: *API call in the browser console*

Creating a multilingual tool for application in NextJS

To add the multi-language possibility to the application we will need the library that called React Intl. Use the following code to add the library to the application:

```
npm i react-intl
// OR
yarn add react-
intl
```

Figure 6.70: *Commands to add the Intl library to the project*

Next, in the root level of the application we will need the folder called “lang” with files that will contain the language translations like this:

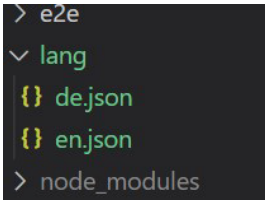


Figure 6.71: *Folder with translations in the application root*

Put this data inside each file depending on the name:

```
// For the de.json
{
    "page.home.head.title": "Kochbuch in Deutsch"
}

// For the en.json
{
    "page.home.head.title": "Cookbook in English"
}
```

Figure 6.72: *Translation strings in the JSON files*

After that, we will need to add configuration to the NextJS. Please add the following code to your NextJS configuration and restart the dev server:

```
const nextConfig = {
  // same code
  i18n: {
    locales: ['en', 'de'],
    defaultLocale: 'en',
  },
  // same code
}

module.exports = nextConfig
```

Figure 6.73: *Updated configuration code for NextJS*

Next, step is to wrap the application with a language detection container. Please update your **__app.page.tsx** file with the following code:

```
// same code
import de from "../lang/de.json";
import en from "../lang/en.json";
import { IntlProvider } from "react-intl";
import { useRouter } from 'next/router';

const messages: any = {
  de,
  en
};

function CookBook({ Component , pageProps }: AppProps ) {
  // same code
  const { locale } = useRouter();
  const localeToString = locale as string;

  return (
    <Provider store={store}>
      <Layout>
        <IntlProvider locale={localeToString}
            messages={messages[localeToString]}>
          <Component {...pageProps} />
        </IntlProvider>
      </Layout>
    </Provider>
  )
}
```

Figure 6.74: *Update _app.page.tsx file*

Finally, we can use the translations in our code. Let us add the string to the main page like this:

```
// same code
import { useIntl } from 'react-intl'
const Home: NextPage = () => {
  const intl = useIntl();

  // same code
  const title = intl.formatMessage({ id: "page.home.head.title" });
  // same code
  return (
    <div className={styles.container}>
      // same code
      <h2>{title}</h2>
      // same code
  )
}
```

Figure 6.75: *Implement the translation string into the page*

Now if we test the application the data on the page will look like this for the English URLs:

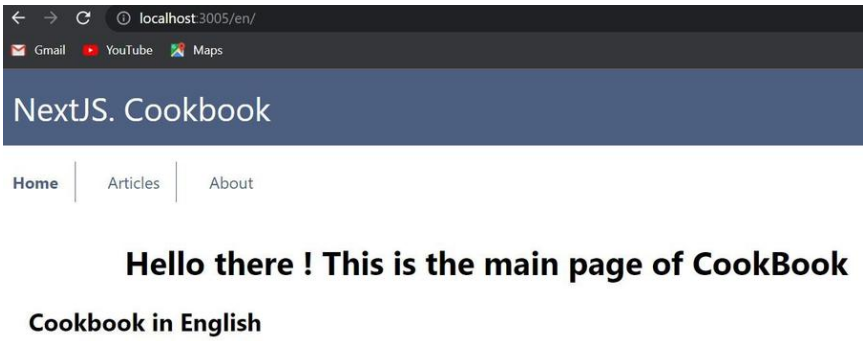


Figure 6.76: *English version of the application*

On the other hand for the German language it will be like the following:

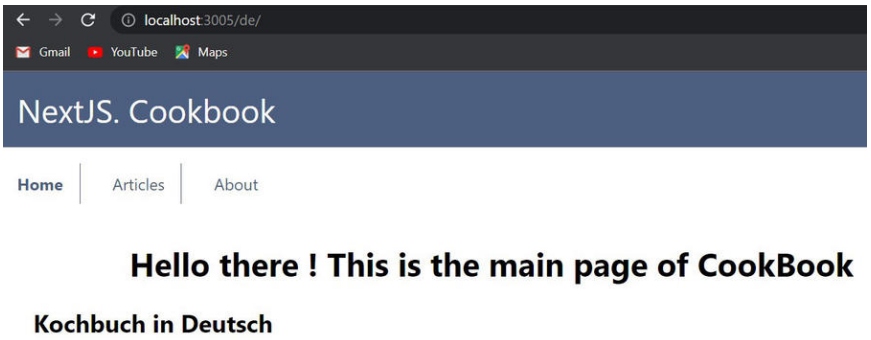


Figure 6.77: *German version on the application*

Conclusion

In this chapter we learned how to implement any internal pages from the very beginning into the application using the test driven development flow. Also, it can be multi-language pages. We can change the application language depending on the user data that can come from API. Please do not stop at the border of the example from this chapter and try to make more pages with more functionality. We will need it in the next chapter where we will implement the E2E testing. That means the more pages we have - the more tests we could create to cover the application. See you in the next chapter.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 7

The superpower of E2E testing in NextJS

Introduction

In the software development world, we know two possible ways of creating the application. The first one - is to create the application just using the requirements and local infrastructure. Second - make an application production ready from the beginning. The first one -is keeping development in the fast lane, but the second one - is keeping the application in the quicker delivery lane. From the business, perspective delivery is what we want at the end of the day. Knowing this we will follow the guides to create the production-ready infrastructure using NextJS as a core framework.

Structure

- Prepare the application for the production release
- Choosing an End-to-End End testing framework
  - Setup Cypress for NextJS
  - Setup Playwright for NextJS
- Writing first e2e test with Playwright

- Creating more tests for the application
  - o Covering the authorization
  - o Covering internal pages
- Conclusion

Objectives

In this chapter, we will learn what is E2E testing, and why it is so essential, for production-ready applications. We will compare the two most popular frameworks that can be used with NextJS. Cover our application with tests and also we cover the authorization of the user.

Prepare the application for the production release

In this, we will create a flow chart that will visualize what environment and infrastructure we need.

In *Figure 7.1* you can find several lanes that represent the development steps from idea to product.

To create the product we will behave as we did from the very beginning of this book. First of all, we will investigate what we want to create and what problem will be solved in the application. Then we will create the requirements plan and wrap it with End-to-End mockups and fake data. After that, we can follow the test driven development rules and create tests for each part of the application to proceed with the code. The coding part - is the most fun part for every software developer. Using all the requirements mocks and tests we can create as much possible error-safe applications.

When we finish with the coding stage we need to test the result somehow. Yes, we have unit testing but, unit tests do not behave as the real user. It can't open the browser and enter the real page. So for this task, we need to create the E2E testing environment and tests. That is what we will learn in this chapter.

Only when we are sure that everything works correctly we can deliver our code into the cloud environment and monitor the product(that is what we will do in the next chapters).

Figure 7.1: *Development flow chart with steps*

Choosing an End-to-End testing framework

There are several solutions on the market that can solve E2E testing problems. The difference is in the possibilities of the platform, the size of documentation the complexity of using and maintaining. As front-end developers, we do not need any complex systems like Selenium or any competitors. But we need some simple tool that will allow us to open the virtual browser and walk through the site step by step. It would be great to use the same Javascript (or Typescript) for it to not learn new languages only for test creation.

Setup Cypress for NextJS

Cypress is a powerful and easy-to-use framework with Chromium support. It has all the required by the developer or QA tools for fast setup, code, and scale of the tests.

To add Cypress to the project, we need to enter the following commands at the root of the project:

```
npm install --save-dev cypress
// OR
yarn add --dev cypress
```

Figure 7.2: *Commands to add Cypress to the project*

Next, we need the command in the **`package.json`** file that will start our Cypress tests. To do it add the command from *Figure 7.3* to file:

```
"scripts": {
  "dev": "next dev --- -p 3005",
  "build": "next build",
  "start": "next start",
  "lint": "next lint",
  "test": "jest",
  "cypress": "cypress open",
  "test:e2e": "yarn playwright test",
  "test:all": "yarn test --coverage && yarn playwright test"
},
```

Figure 7.3: *Command that will start the Cypress tests*

After the first run, you will see the debug window like in *Figure 7.4:*

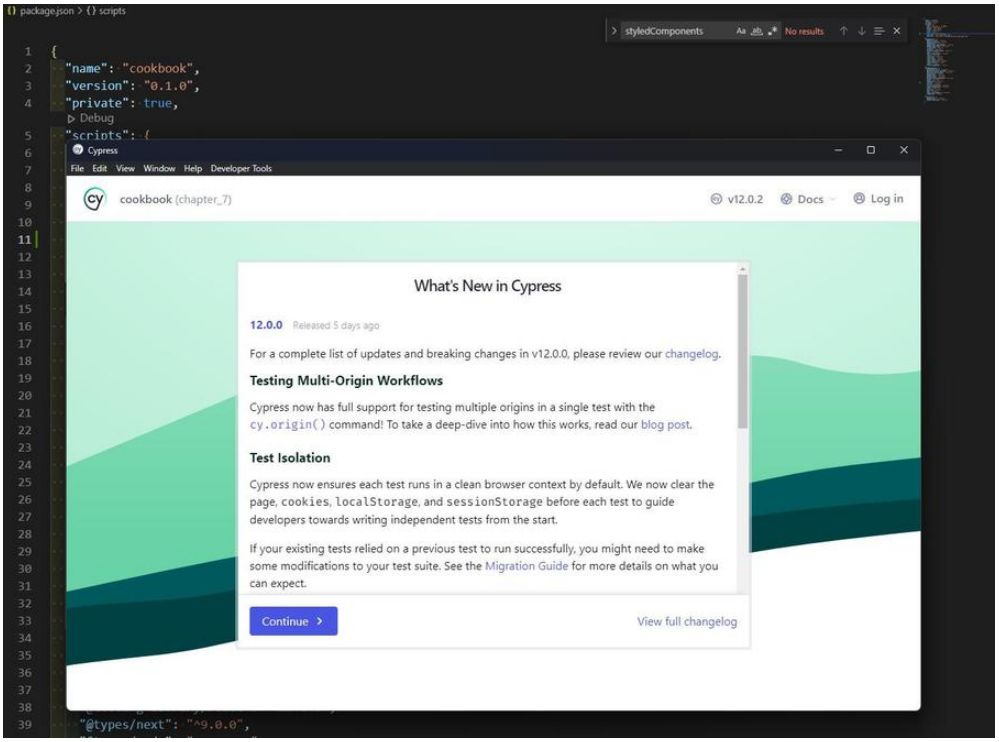


Figure 7.4: *First run of Cypress*

Then you will see this window:

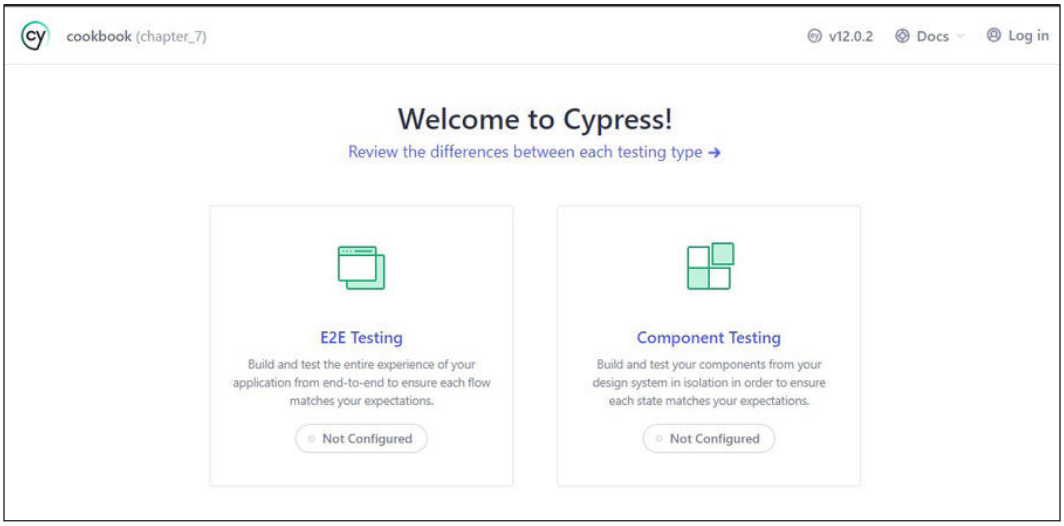


Figure 7.5: *Start screen for Cypress*

Let us review the difference of choice provided by Cypress:

E2E tests are required compiled version of the project and will follow the URL links the same way as any real user will. That means that we will have pages that contain all components in the browser. The tests will contain checks that the user will rightly see on the page:

Component tests it is the same tests that we did before using Jest. These tests will use isolated components and contain checks for the components only. That means if we choose Cypress as a testing framework we could get rid of Jest and use Cypress as the only testing framework

You can also see the different descriptions provided by Cypress in *Figure 7.6:*

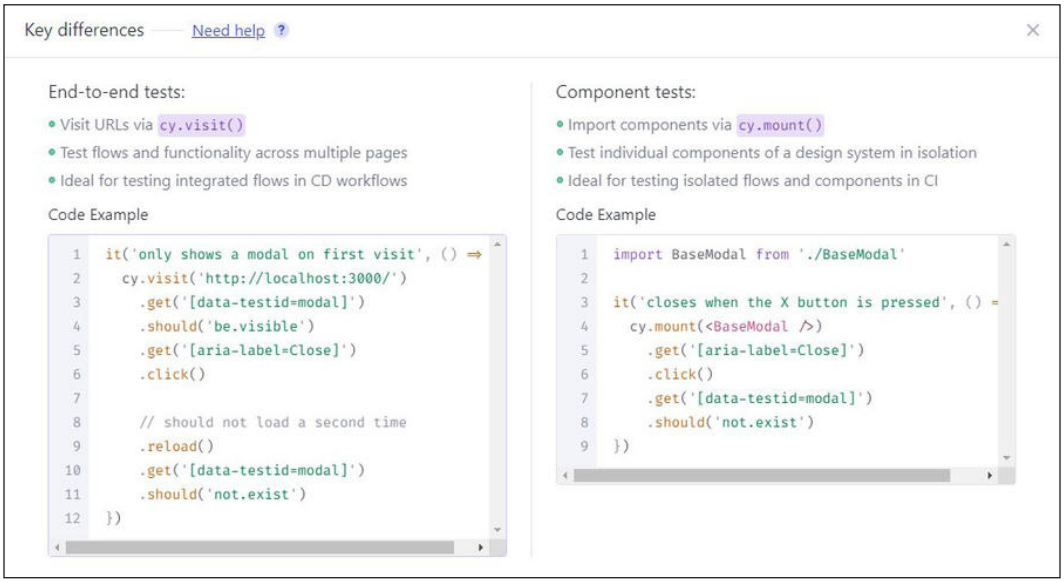


Figure 7.6: *Description provided by Cypress*

Click on the left block from *Figure 7.6* to configure the E2E tests using Cypress. After clicking, you will see that Cypress added some files to the project like in *Figure 7.7:*

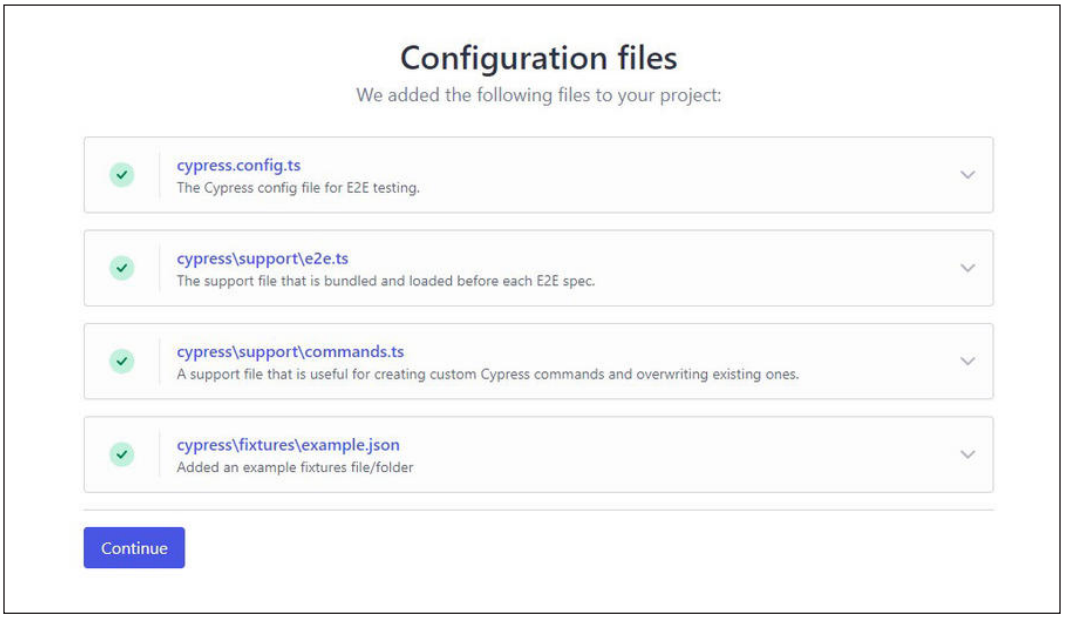


Figure 7.7: *Configuration result for E2E using Cypress*

In the next step you will see the screen where you will need to choose the browser to test (you could also check the Electron version of the project):

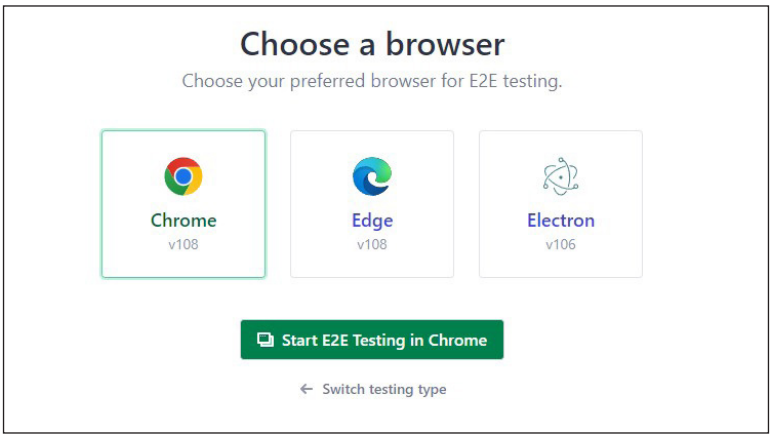


Figure 7.8: *Browser configuration in Cypress*

After you click "**`Start E2E testing in Chrome`**" (It is chosen in Chrome but you can choose any other preference), Chrome browser will open the new controlled window with content provided in *Figure 7.9:*

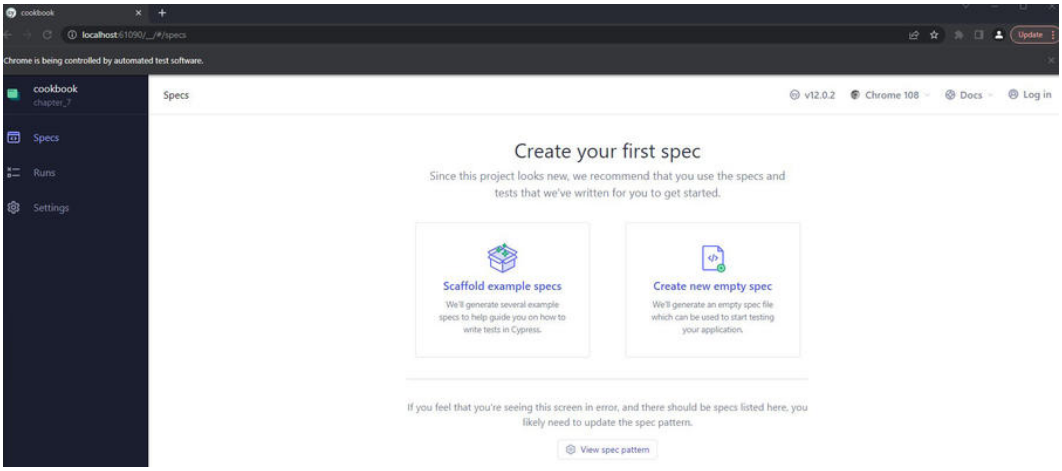


Figure 7.9: *Cypress testing interface*

Now you can create an empty test by choosing a right block, or scaffold example using a left block. We will choose the left one for example purposes. You will see the content on your page that contains the list of example specifications that was added to the project as depicted in *Figure 7.10:*

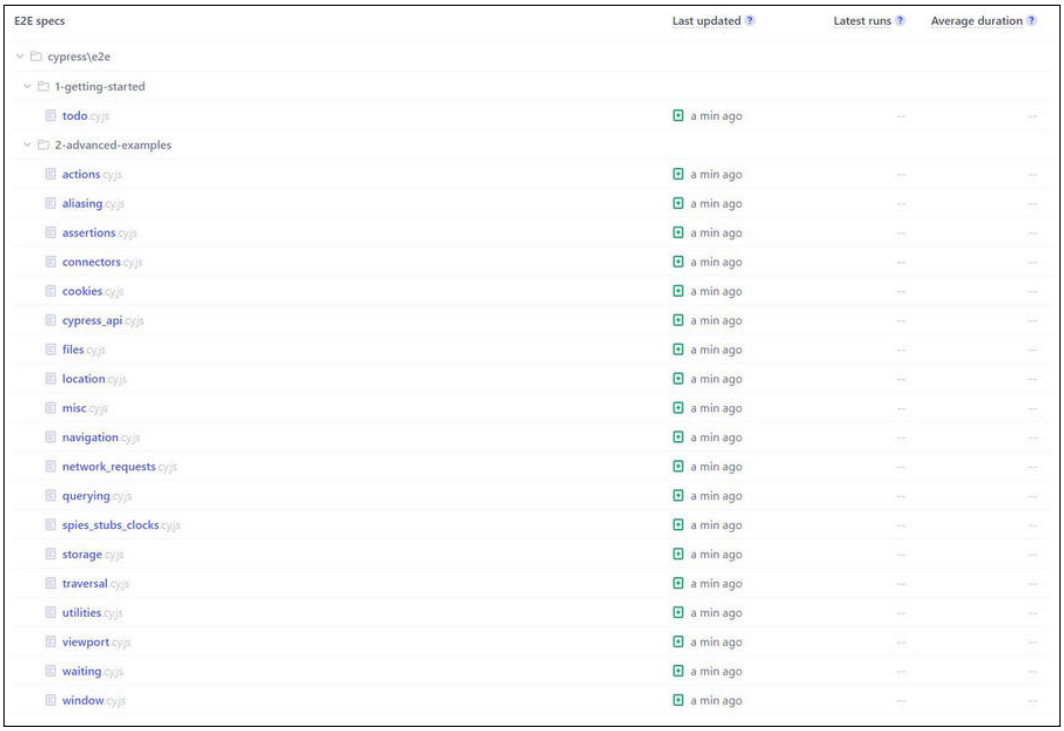


Figure 7.10: *Example specifications provided by Cypress*

In this list, you can click on each test that will be run immediately. Let us click on **`todo.cy.js`** file for example.

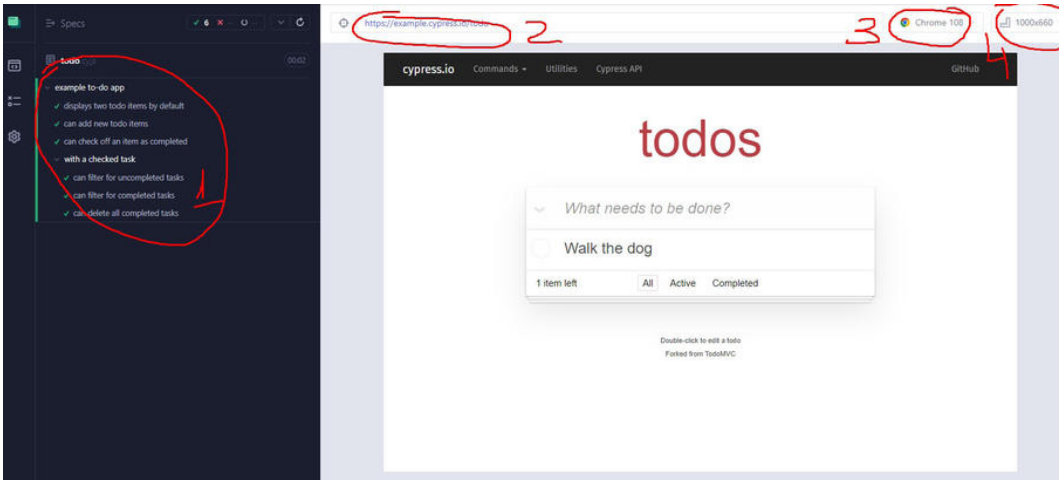


Figure 7.11: *E2E test result*

After clicking you will see the process of the E2E testing. Cypress will follow the instructions provided in section 1 in *Figure 7.11*. It will open the URL from section 2 . The browser that will be used for the test is in section 3 in *Figure 7.11*. Section 4 is the resolution of the screen that will be used for the test. This configuration is controllable and can be configured before we start any test.

If you open the test instruction from section 1 in *Figure 7.11* and hover over one of the instructions you will also see the part of the page that is tested by this instruction like in *Figure 7.12:*

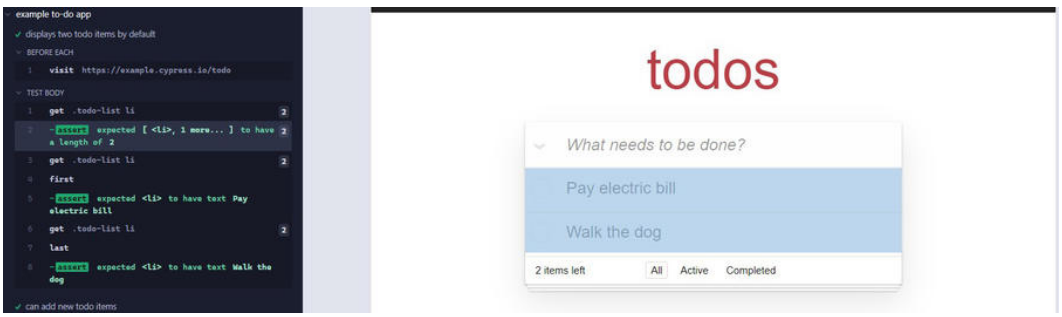


Figure 7.12: *Example of hover behavior*

We can also check the state change of elements. For example, we have the checkbox state change and need to figure out the difference before and after action. To get the information we can click on the action in the list and then we can see the action box

that can show us the difference. You can click on the "**`before`**" and "**`after`**" buttons to change the state. The result you can see in *Figure 7.13:*

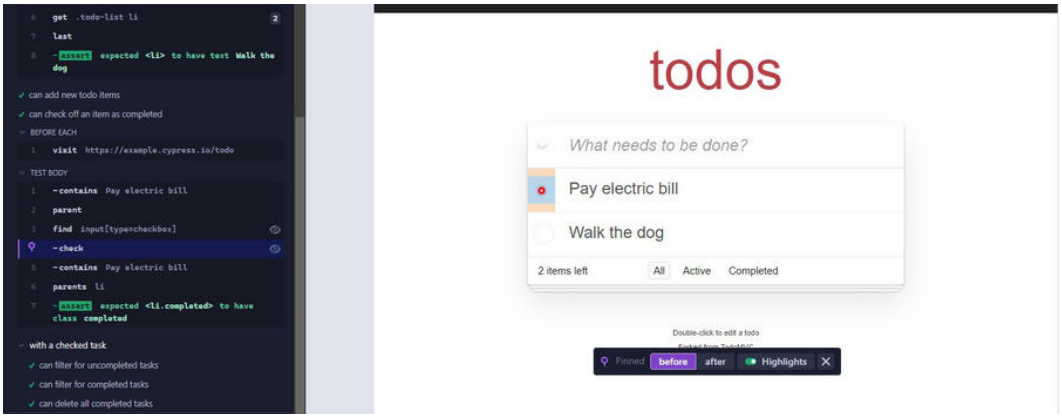


***Figure 7.13**: The state change for the checkbox element*

By default configuration, the IDE for the project will be VS Code if you will not change this parameter. If you click on the "**`Before each`**" or "**`Test body`**" section there will be a button to open the test code in the IDE like in *Figure 7.14*.

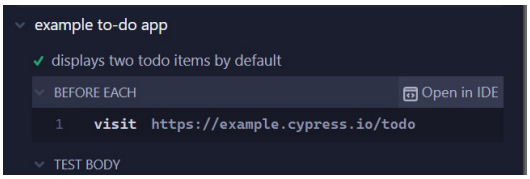


***Figure 7.14:** Open in IDE button for the test*

Inside the file, you can see regular E2E test code like in *Figure 7.15:*

```
describe('example to-do app', () => {
  beforeEach(() => {
    // Cypress starts out with a blank slate for each test
    // so we must tell it to visit our website with the `cy.visit()` command.
    // Since we want to visit the same URL at the start of all our tests,
    // we include it in our beforeEach function so that it runs before each test
    cy.visit('https://example.cypress.io/todo')
  })

  it('displays two todo items by default', () => {
    // We use the `cy.get()` command to get all elements that match the selector.
    // Then, we use `should` to assert that there are two matched items,
    // which are the two default items.
    cy.get('.todo-list li').should('have.length', 2)

    // We can go even further and check that the default todos each contain
    // the correct text. We use the `first` and `last` functions
    // to get just the first and last matched elements individually,
    // and then perform an assertion with `should`.
    cy.get('.todo-list li').first().should('have.text', 'Pay electric bill')
    cy.get('.todo-list li').last().should('have.text', 'Walk the dog')
  })
```

***Figure 7.15:** Example of E2E code from Cypress*

Setup playwright for NextJS

The next framework is the outstanding result of the Microsoft team that can use any browser in test flow, authorize once by test, test API, and many other features.

We have added the Playwright to our project in the previous chapters. But if you missed this part please follow the steps above.

To connect Playwright you need first enter these commands from *Figure 7.16* in the console:

```
npm install --save-dev @playwright/test
// OR
yarn add --dev @playwright/test
```

Figure 7.16: *Commands to add playwright*

You will also need this command in the **`package.json`** file to run the playwright:

```
"test:e2e": "yarn playwright test",
```

Figure 7.17: *Command in the package.json file to run Playwright*

After the installation Playwright will add an example specification file that we could use to check the possibilities of the framework. Let us run the command to check the result of it: (*Figure 7.18*)

```
Running 75 tests using 8 workers

  75 passed (60s)

To open last HTML report run:

  npx playwright show-report

Done in 60.82s.
```

Figure 7.18: *Playwright command using the result*

As you can see from *Figure 7.18* we passed all provided in the example tests and also see the command to see the report. Let us enter this command to see the report of the tests. The result can be seen in *Figure 7.19:*

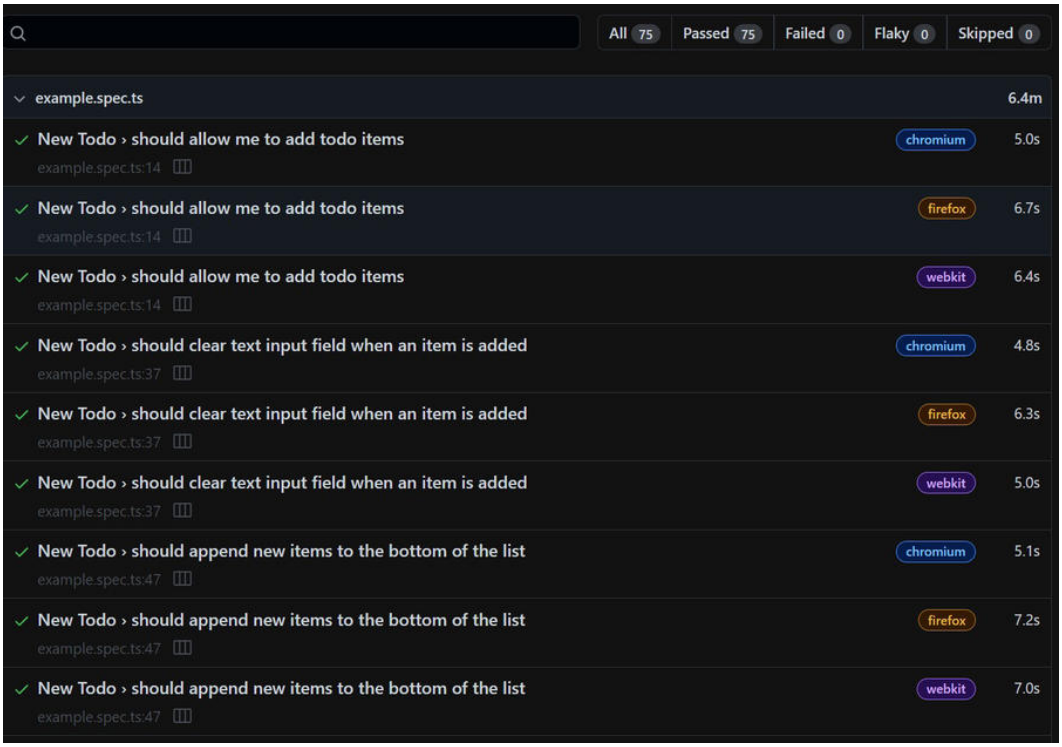


Figure 7.19: *Report of Playwright tests*

In this report (from *Figure 7.19*) you can get this information:

- What test instruction was loaded
- In what browser
- Time of operation
- The file name of the instruction
- Status of the test instruction

In the next step, we can click on instructions and check the report of exact instructions. The result of it can observe in *Figure 7.20:*

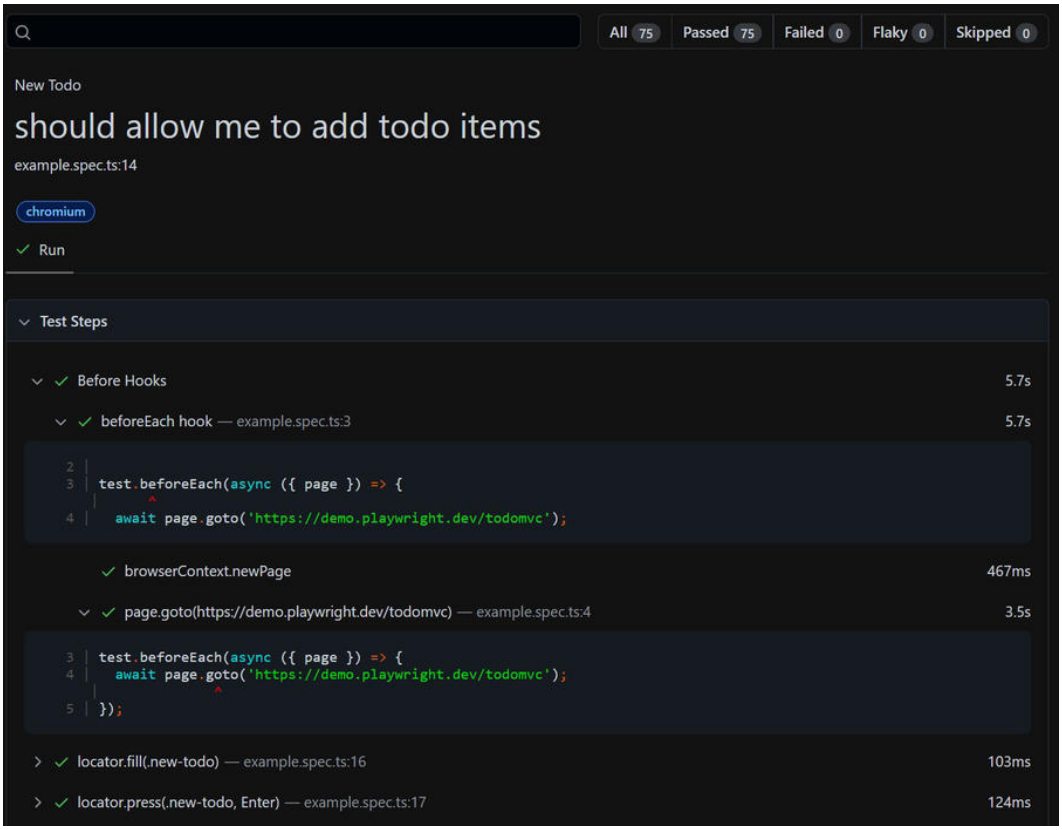


Figure 7.20: *Report for the selected instruction*

Now (in *Figure 7.21*), we can see the full report that was done in the instruction line by line with the status of execution.

Let us generate the error to see what will be if the test will fail. You can see it in *Figure 7.21*:

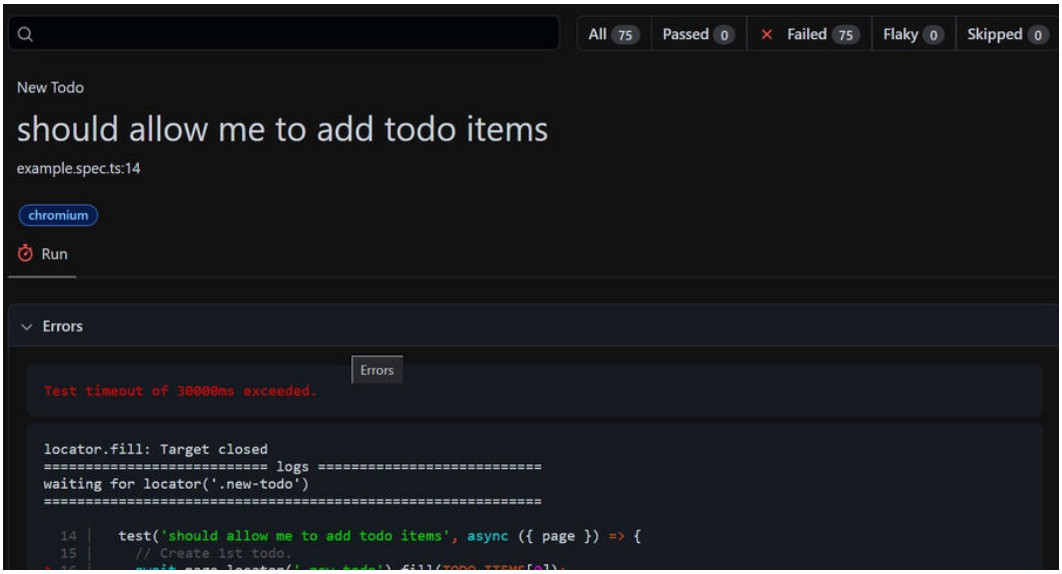


Figure 7.21: *Failed test example in Playwright*

We have made the test to call the wrong URL so the error report shows that timeout was exceeded and the robot could not open the page. Next what we can see here - is the line and what instruction was called to generate the error in this test. Also at the top information line, you can see the number of failed tests.

Next what we need to know about Playwright - is the possibility to check the traces. Traces is the same information that was in the browser at the moment of execution. That means that we could see the console.log and network information at the execution time. To get this information we need to call the Playwright test with a special flag like this:

```
npx playwright test --trace on
```

Figure 7.22: *Run the test with the trace-on flag*

After that, you will see an updated report in your browser. There will be a new section with a trace file and trace view like in *Figure 7.23:*

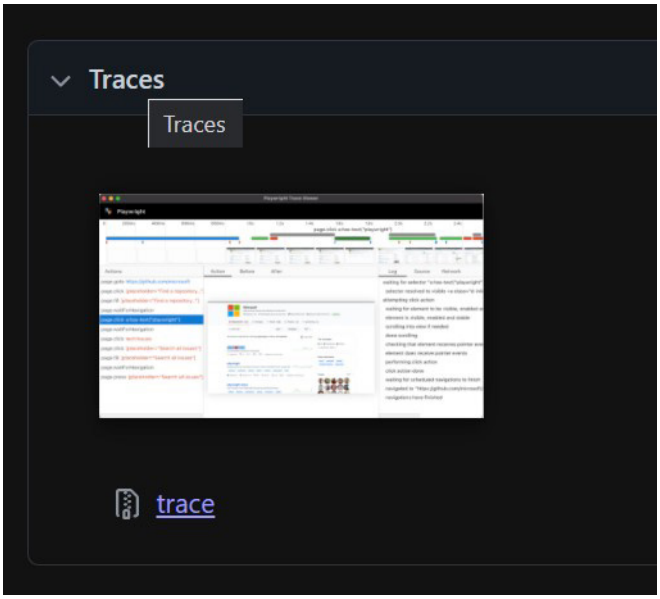


Figure 7.23: *Traces functionality in the Playwright Report*

By clicking on the picture, we will be led to the traces page where we can see the whole history second by second what was in the screen, console, network, and metadata like in *Figure 7.24:*

Figure 7.24: *Traces view for the test*

We can follow the robot's actions second by second and see what data was sent, what consoles were on the browser, what steps were correct or not, and the like. This trace view will give us full information about what was wrong with the page. Also, in *Figure 7.25* there is a link to download the archive. This archive contains all the trace information that can be integrated into your internal monitoring system and provides the trace view there.

Writing the first e2e test with Playwright

As you can guess, we will choose Playwright as a testing framework for our application. Let us try to create the first test for our application using the example specifications.

For more convenient use of Playwright in your IDE please install this extension from *Figure 7.25:*

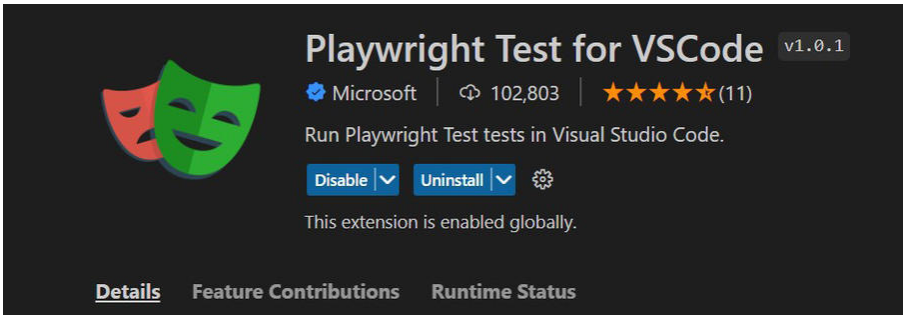


Figure 7.25: *Extension for the VSCode to use Playwright tests in IDE*

After that you will see the update in your test area in the IDE like this:

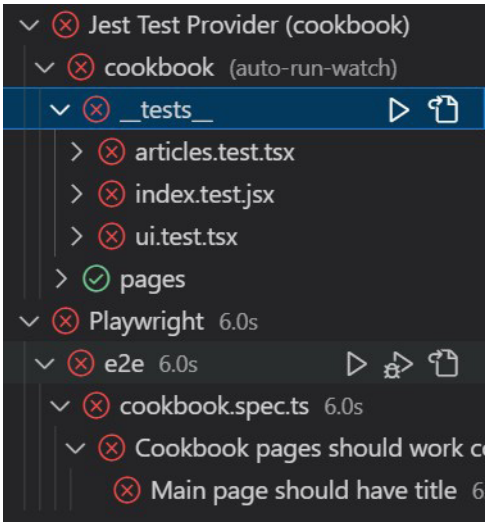


Figure 7.26: *Updated test view in the VSCode*

In my example, we have already added the first test into the test flow. To add it on your side please create the file with the name **`cookbook.spec.ts`** in the e2e folder (do not forget to remove or rename existing example files in this folder). Also, we will require some configuration for the tests so please create the configuration file too.

The file structure example is in *Figure 7.27:*

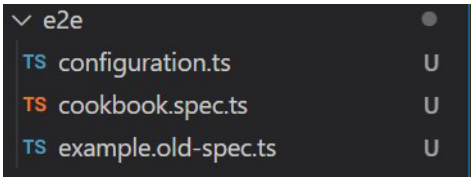


Figure 7.27: *Example file structure*

In the newly created file please add this code from *Figure 7.28:*

```
import { test, expect, Page } from '@playwright/test';
import { Configuration, expectations } from './configuration';

test.beforeEach(async ({ page }) => {
    await page.goto(Configuration.HOST);
});

test.describe('Cookbook pages should work correctly', () => {
    test('Main page should have title', async ({ page }) => {
        await page.goto(Configuration.HOST);
        await expect(page.locator('h1')).toHaveText(expectations.mainPage.header);
    });
});
```

Figure 7.28: *First e2e test with Playwright*

And in the configuration, there should be this code as in *Figure 7.29:*

```
enum Configuration {
    HOST = 'http://localhost:3005'
};

const expectations = {
    mainPage: {
        header: 'Welcome to the cookbook'
    }
}

export { Configuration, expectations };
```

Figure 7.29: *Configuration for the e2e test*

Now if you check the visual part of the test (or if you run the test) you will see that it is red and failed. It is because the text in the h1 tag is not fit the expectations. We

need to fix it and add the text to the h1 tag. After that, you will see that all tests are green.

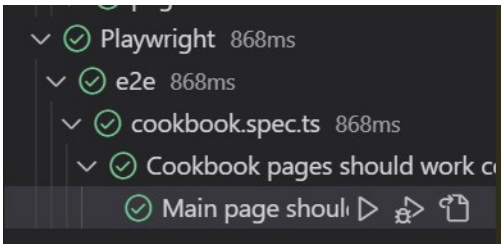


Figure 7.30: *Green tests for Playwright*

Creating more tests for the application

As our application uses more than one page we need to create more tests. We need to create these tests:

- Main page (already have it)
- Articles list page
- Article page
- Login page

Covering the authorization

We are using authorization in the application, so we will require to add authorization into the flow. There are several ways to achieve it using Playwright. We will take the most basic one and add the entering login page and authorization before each test.

To create the login flow add this code into the before each section:

```
test.beforeEach(async ({ page }) => {
    await page.goto(Configuration.HOST + '/login');
    await page.locator('#login').fill(expectations.auth.login);
    await page.locator('#password').fill(expectations.auth.password);
    await page.locator('#submit-login').click();
});
```

Figure 7.31: *Authorization flow in before-each section*

After that each time when you start your test, the robot will open the login page and do authorization and you can expect the elements on the page that be shown only for authorized users. Add this code to your test to check if we have an add-article button:

```
test('Articles list should show data',async ({page}) => {
    await page.goto(Configuration.HOST + '/articles');
    await expect(page.locator('button').first()).toHaveText([
        expectations.articles.add
    ]);
})
```

Figure 7.32: *Test for the article page to check the button*

We also need to update the configuration like in *Figure 7.33:*

```
import { getMock } from "../pages/mocks";

enum Configuration {
    HOST = 'http://localhost:3005'
};

const expectations = {
    mainPage: {
        header: 'Welcome to the cookbook'
    },
    auth: {
        login: getMock.users[0].user,
        password: getMock.users[0].password
    },
    articles: {
        add: '+ add article'
    }
}

export { Configuration, expectations };
```

Figure 7.33: *Updated configuration file*

Please find the updated test code in *Figure 7.34:*

```
import { test, expect, Page } from '@playwright/test';
import { Configuration, expectations } from './configuration';

test.beforeEach(async ({ page }) => {
    await page.goto(Configuration.HOST + '/login');
    await page.locator('#login').fill(expectations.auth.login);
    await page.locator('#password').fill(expectations.auth.password);
    await page.locator('#submit-login').click();
});

test.describe('Cookbook pages should work correctly', () => {
    test('Main page should have title', async ({ page }) => {
        await expect(page.locator('h1')).toHaveText([
            expectations.mainPage.header
        ]);
    });
    test('Articles list should show data',async ({page}) => {
        await page.goto(Configuration.HOST + '/articles');
        await expect(page.locator('button').first()).toHaveText([
            expectations.articles.add
        ]);
    })
})
```

Figure 7.34: *Full code of updated E2E test*

Covering internal pages

Let us add some more tests for internal pages to have better coverage. We will follow this flow to create the tests for the articles:

- Articles list should show the data list
- In the article list if we click on the edit button modal window should be displayed
- If we click add article button then also modal window should be displayed
- On the article, the page should be data that exist in the article

To cover these requirements please update the test file with the following code from *Figure 7.35:*

```
test('Articles item should show data',async ({page}) => {
    await page.goto(Configuration.HOST + expectations.articles.itemUrl);
    await expect(page.locator('h1')).toHaveText([
        expectations.articles.itemHeader
    ]);
})
test('Add article should open the modal',async ({page}) => {
    await page.goto(Configuration.HOST + '/articles');
    const buttons = await page.locator('button').first();
    buttons.click();
    await expect(page.locator('id=edit').first()).toBeVisible();
});

test('Edit button should open the modal',async ({page}) => {
    await page.goto(Configuration.HOST + '/articles');
    const editButtons = await page.locator('role=checkbox');
    editButtons.first().click();
    await expect(page.locator('id=edit').first()).not.toBeHidden();
});

test('Article page should render the page',async ({page}) => {
    await page.goto(Configuration.HOST + '/articles');
    const editButtons = await page.locator('role=checkbox');
    editButtons.first().click();
    await expect(page.locator('id=edit').first()).not.toBeHidden();
})
```

Figure 7.35: *Tests that will cover the requirements*

Please note that we need to use `await` for each action because this directive creates the wait-for-render mechanism that safely wait that element is rendered on the page.

We will also require to update the expectations array to have the mock data. Collect the code in *Figure 7.36:*

```
const expectations = {
    mainPage: {
        header: 'Welcome to the cookbook'
    },
    about: {
        header: 'This is the About page of CookBook'
    },
    auth: {
        login: getMock.users[0].user,
        password: getMock.users[0].password
    },
    articles: {
        header: 'Articles list',
        add: '+ add article',
        itemUrl: '/articles/123/',
        itemHeader: 'test title'
    }
}
```

Figure 7.36: *Updated expectations array*

As you see, we added the role identification to the button. This is needed to call this button by identification. You can use anything that will be convenient for you (class, role, data). In this book, the button is used as the checkbox for the modal window so we could add **`role=checkox`** for it.

Now if you use the extension for the VSCode you will see that all tests are green. That means that all tests are green as in *Figure 7.37:*

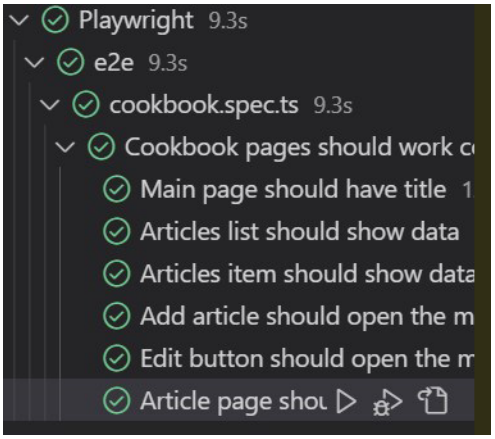


Figure 7.37: *Result of Playwright tests*

Conclusion

This summary effectively wraps up the chapter on end-to-end (E2E) testing, highlighting the benefits of incorporating E2E tests into a developer's daily workflow. By using E2E tests, developers can reduce the number of cycles required before moving to production, and E2E tests can also be integrated into CI/CD pipelines, which is particularly valuable for larger teams and projects

The next chapter will lead us to the most important part of the development. We will introduce the delivery stage and deployment to the production. To achieve it we will check the best possible on-market tools that you can use for your next project. See you in the next chapter.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 8
Deploying NextJS project to production

Introduction

In modern software development, the creation and coding stage is only part of the lifecycle of the development process. As you can remember from the last part it is only a tiny piece of the process that is surrounded by a big number of steps. One of the important parts is delivering software to the end user. On the web, it is the deployment of applications into the cloud service, hosting, or any other servers. We will cover the topic of cloud services in this chapter as it is the most modern way of using the web today.

Structure

- Preparing the project to fly into production
  - Choosing the "perfect" render for the application.
  - Measuring performance and maintainability applications in NextJS.
  - Connecting Sentry for application monitoring.
- Using AWS Amplify to host our application
  - Understanding Amplify Admin area

    - Creating the data models for the application
    - Creating an authentication flow with AWS
- Adding data in the admin area
- Using cloud functions for application
- Reuse Cloud Functions with Layers functionality
- Finishing the backend with Amplify
- Host the application in the cloud and first run
- Conclusion

Objectives

In this chapter, we will speak about cloud hosting solutions and especially AWS Amplify. This service from AWS is covering all possible quick-start requirements of cloud hosting and service. Amplify is containing a visual backend service that provides GraphQL as the result (this is why we covered this topic in previous chapters), also there is a connection to any GIT service and build pipelines. The service itself is partly free which will allow anyone to have a quick launch of your idea in a very short time.

Preparing the project to fly into production

Before we start to deploy any code base into the cloud we need to speak about what exactly we will send into web hosting. Regarding NextJS there are several ways to the build result from the raw code into a complete ready-to-use application. For development purposes, we start a live server that allows us to see the hot updates immediately but in production, this is bad practice so we need to build the code into the application before use.

Choosing the "perfect" render for the application

Let us speak about the possible way to create the user-end application in NextJS. Regarding the framework in our toolbelt, there are several build types provided:

- **Client-Side rendering (CSR)**: In this case, the whole project will be the same Single-Page application as any regular React application. In this way, you will get the HTML page that will fetch the data dynamically so at the first

load the page itself will be super-light and loads as fast as possible. This is a piece of good news. The bad news is that for SEO this way of rendering is one big disadvantage. Robots will sync your data less effective and will require more attention with search engines. The ranking of your pages will be lower than the optimized ones that generated on the server. This way of project build can be recommended for the internal systems, that do not require any actions from the search engine robots and are used as a business system. As an example, we can take CRM systems, dashboards, or internal messaging.

- **Incremental Static Regeneration (ISR)**: In this case, pages will be generated on the fly without the whole project being built. This way of rendering is very interesting as its fits the SEO requirements, and also does not need the update of the project each time you add the page to the project. That means, for example, you have a generatable page like **[pid].page.tsx,** and each time you create the new PID for this page there is no need to rebuild the project, and the old pages will be cached. It could look like the perfect build for the project but there are also disadvantages. On the other hand - we will require more server resources depending on the number of pages and users that could start growing.
- **Server Side Rendering (SSR):** This way is pretty much the same as ISR but the difference is that the page will be re-rendered on the server side for each request. In this way, we will need even more resources compared to ISR.
- **Static Site Generation (SSG):** In this case, HTML will be generated in the build stage. In this way, we can store and cache files in CDN for performance increase. We can also choose between static site generation with fetched data or without. The disadvantage of this way is that each time we add a page or make changes in the content we need to rebuild the whole project. But an advantage is the performance of the project in production.

To choose what build type is perfect for your next project please answer several questions to help yourself:

- How many times per period you will add or update pages and content of the site?
- What hosting provider do you have for the project?
- Do you need SEO?
- Do you need a cache for each page or only some of them?

The answers will help you to recognize what kind of project you have and how to choose your perfect way of building and delivery.

Measuring performance and maintainability applications in NextJS.

Before we speak about performance measuring of the application, we need to implement some basic knowledge about 'what is performance' generally and how to measure it on the web.

To solve this issue we will add here information about Web Vitals as it is the base of the performance measuring in the web industry. Web Vitals is the initiative by Google that provides the signals to help improve your web application. Metrics of the Web Vitals are separated into three different types:

- **Largest Contentful Paint (LCP):** this type will measure application loading performance. The average time of the application loading is about 2.5 seconds after the page first start. This number is provided by the Web Vitals rules.
- **First Input Delay (FID):** this type will measure interactivity. The delay should be no more than 100 milliseconds to fit the good user experience rule.
- **Cumulative Layout Shift(CLS)**: this type will measure visual stability in scores. The score itself is measuring the difference between the page before action and after (for example asynchronously loaded content after the page is loaded). The good score should be 0.1 or less.
- **Time to First Byte (TTFB)**: this type shows the time between the browser requesting a page when the first byte was received from the server. That means that we take the time between the request start and response start in milliseconds.
- **First Contentful Paint (FCP)**: this type measures the time from when the page starts loading to when any element of the page is rendered on the screen. The average time for a good load is about 1.8 sec.
- **Interaction to next Paint (INP)**: this metric is assets responsiveness and causes when the page becomes unresponsive. This metric is experimental so do not expect strict metrics values.

In *Figure 8.1* we can observe visually how the types work:

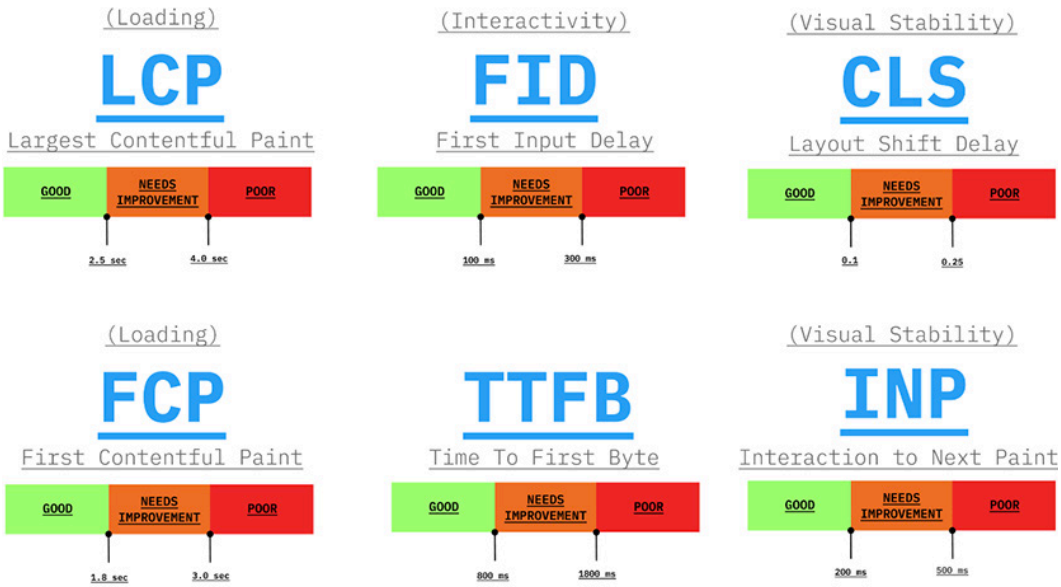


Figure 8.1: *Web Vitals visually*

Directly, we can implement this measure in our application using internal tools that exist in NextJS. Add this code in **`_app.page.tsx`** to see the performance data in your browser console as shown in *Figure 8.2*:

```
// Same code
export function reportWebVitals(metric: NextWebVitalsMetric) {
  console.log(metric)
}

function CookBook({ Component , pageProps }: AppProps ) {
    // Same code
}

export default CookBook
```

Figure 8.2: *Function to get*

After the page reloads, we will see the data in the browser console like in *Figure 8.3:*

Figure 8.3: *Metrics in the browser console after code update*

As you can see, we can filter the metric data by the label, to get only web-vital information or by the name to separate each metric.

Also, there is a NextJS-specific metrics that exist in the report on the page load or route change. These metrics will have a custom label and can be separated by the name:

- **Next.js-hydration**: The time of hydration in milliseconds
- **Next.js-route-change-to-render**: The time that takes to start rendering the page after router change

- **Next.js-render**: The time that takes to finish rendering the page after a route change

In *Figure 8.4* we can see the example of a report in the browser console after the route change:

Figure 8.4: *Custom NextJS metrics*

Getting metrics is only half of the maintainability process. We need to send this data somewhere. As you can see the data itself is just an object so we can use any API to collect the data. To send it into the API we can use the code example from *Figure 8.5:*

```
export function reportWebVitals(metric: NextWebVitalsMetric) {
  const body = JSON.stringify(metric)
  const url = 'https://example.com/analytics'

  // Use `navigator.sendBeacon()` if available, falling back to `fetch()`.
  if (navigator.sendBeacon) {
    navigator.sendBeacon(url, body)
  } else {
    fetch(url, { body, method: 'POST', keepalive: true })
  }
}
```

Figure 8.5: *Send metrics example code*

For modern browsers, there is a **`sendBeacon`** function that asynchronously sends HTTP Post requests containing a small amount of data to some web server. If there is no such function we will use regular fetch (that is not recommended). As a receiving data API, we can use any service that provides this possibility (for example: Elastic Kibana, Zabbix, or Dynatrace). In the example from *Figure 8.6* we will use the Google Tag manager to store the browser metrics:

```
export function reportWebVitals({ id, name, label, value }) {
  window.gtag('event', name, {
    event_category:
      label === 'web-vital' ? 'Web Vitals' : 'Next.js custom metric',
    value: Math.round(name === 'CLS' ? value * 1000 : value), // values must
be integers
    event_label: id, // id unique to current page load
    non_interaction: true, // avoids affecting bounce rate.
  })
}
```

Figure 8.6: *Example of using Google Tag to store the metrics*

Connecting Sentry for application monitoring

Before we start implementation let us speak about the service itself and why we choose it as the performance monitoring tool. The main motivation is that the service was created as a tool to catch JavaScript errors in the project. Today this feature is only a small part of the big ecosystem. That means we could wrap our project not only with the solution that will help us monitor the project's performance data but also any errors that could happen while using the app. And the main reason, that Sentry is the ready-to-use ecosystem with dashboards and notifications. These possibilities are not in the built-in scope for NextJS so we will proceed with Sentry.

First, open the **`Sentry`** in the browser using this link **`https://sentry.io/`.** In *Figure 8.7* you can see the main page:

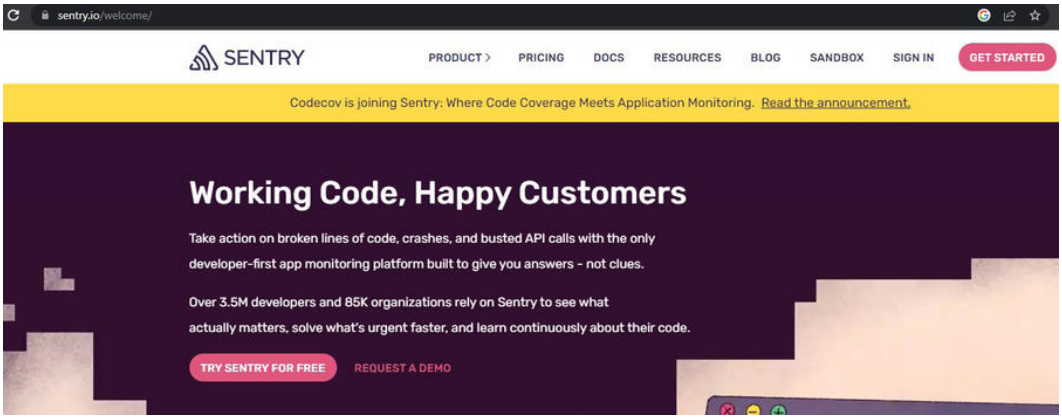


Figure 8.7*: Sentry main page*

Then you need to click the **`Get started`** button to proceed. In the registration form please fill all required data or choose Sign-in with social media(I will choose sign-in with Google):

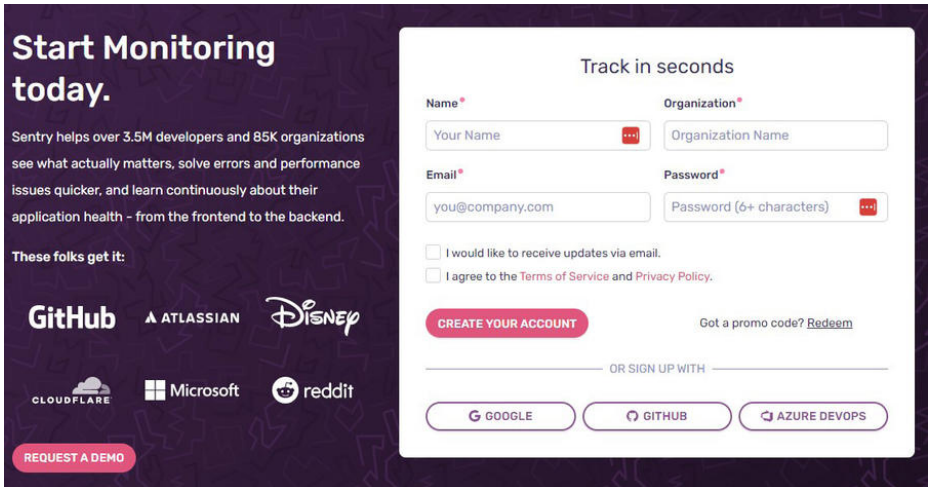


Figure 8.8: *Registration form for the Sentry*

Now you can create a new project for the implementation by clicking on the **`Create project`** button.

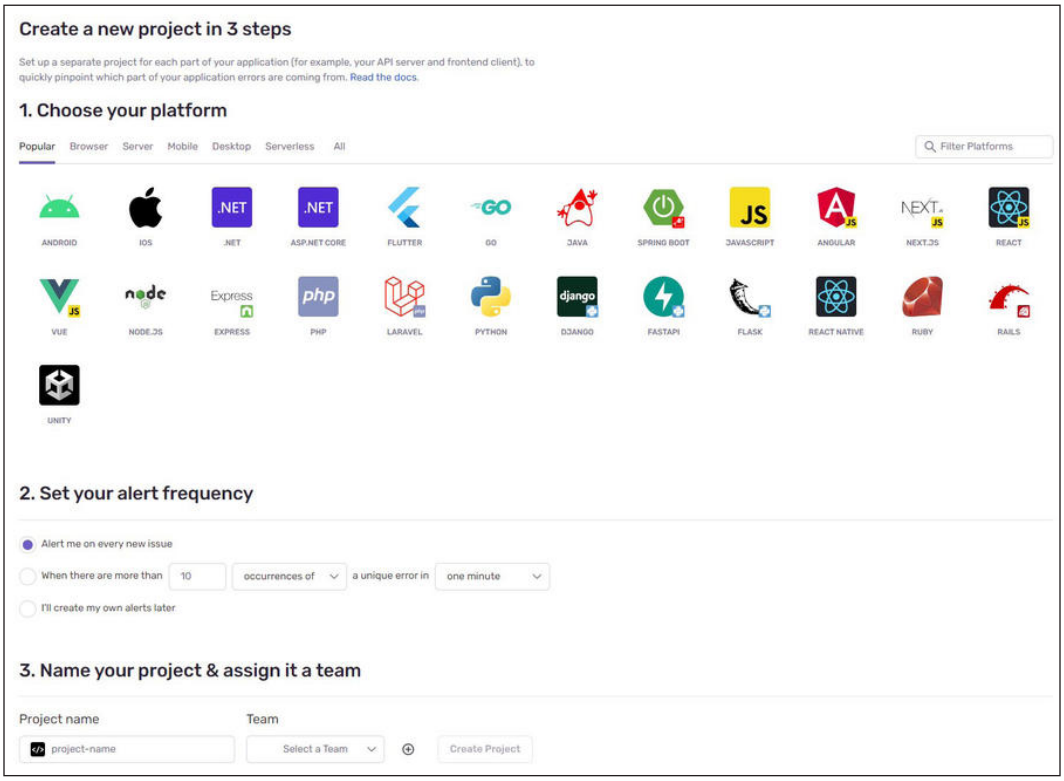


Figure 8.9*: Creating a new project in Sentry*

In the screen from *Figure 8.9*, we need to the setup project. As you can see we can choose NextJS as a platform. We will keep an alert frequency for every issue by default. The name and team of the project will be **CookBook** as well. When everything is filled we can press the `Create project` button.

In the next step, we will get instructions on how to implement Sentry in our project. Please use these commands from *Figure 8.10* to install Sentry in our project:

```
yarn add @sentry/nextjs
# or
npm install --save @sentry/nextjs
```

Figure 8.10: *Commands to install Sentry in the project*

After that, we can configure the Sentry. Use this command to achieve it:

```
npx @sentry/wizard -i nextjs
```

Figure 8.11: *Command to configure Sentry in the project*

As we are using TypeScript in the project so we need to do some manual changes to activate sentry in the project. Sentry setup will create several files automatically. We need to change the names of these files. Check *Figure 8.12* to see the file structure that will be changed:

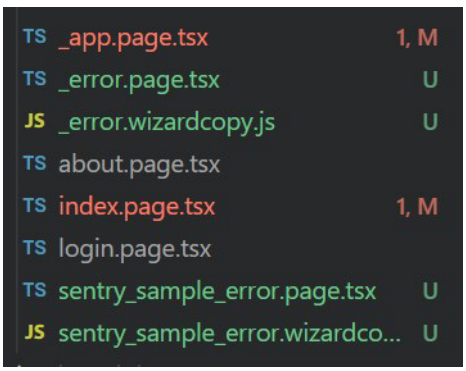


Figure 8.12: *Files structure to be changed*

As you see we will rename **`_error.js`** and **`sentry_sample_error.js`** files to **`_error.page.tsx`** and **`sentry_sample_error.page.tsx`.** We will need to make changes in the **`next.config`** file. Please use *Figure 8.13* to make updates:

```
const { withSentryConfig } = require("@sentry/nextjs");

const nextConfig = {
  // Same code
  sentry: {
    silent: true,
  },
   // Same code
}

module.exports = withSentryConfig(nextConfig);
```

Figure 8.13: *NextJS configuration update*

Finally, we need to check if everything works correctly. To do that let us add a button that will trigger the error on click. This error will be sent into Sentry automatically. Update your index page as provided in *Figure 8.14:*

```
// Same code
const Home: NextPage = () => {
  // Same code
  return (
     // Same code
      <button
          type="button"
          onClick={() => {
            throw new Error("Sentry Frontend Error");
          }}
        >
          Throw error
        </button>
   // Same code
  )
}

export default Home
```

Figure 8.14: *Home page update to send error into Sentry*

When everything is configured you can restart the server and try to click the button. When you trigger the error in the Sentry console you will see a new issue generated in the list like in *Figure 8.15:*

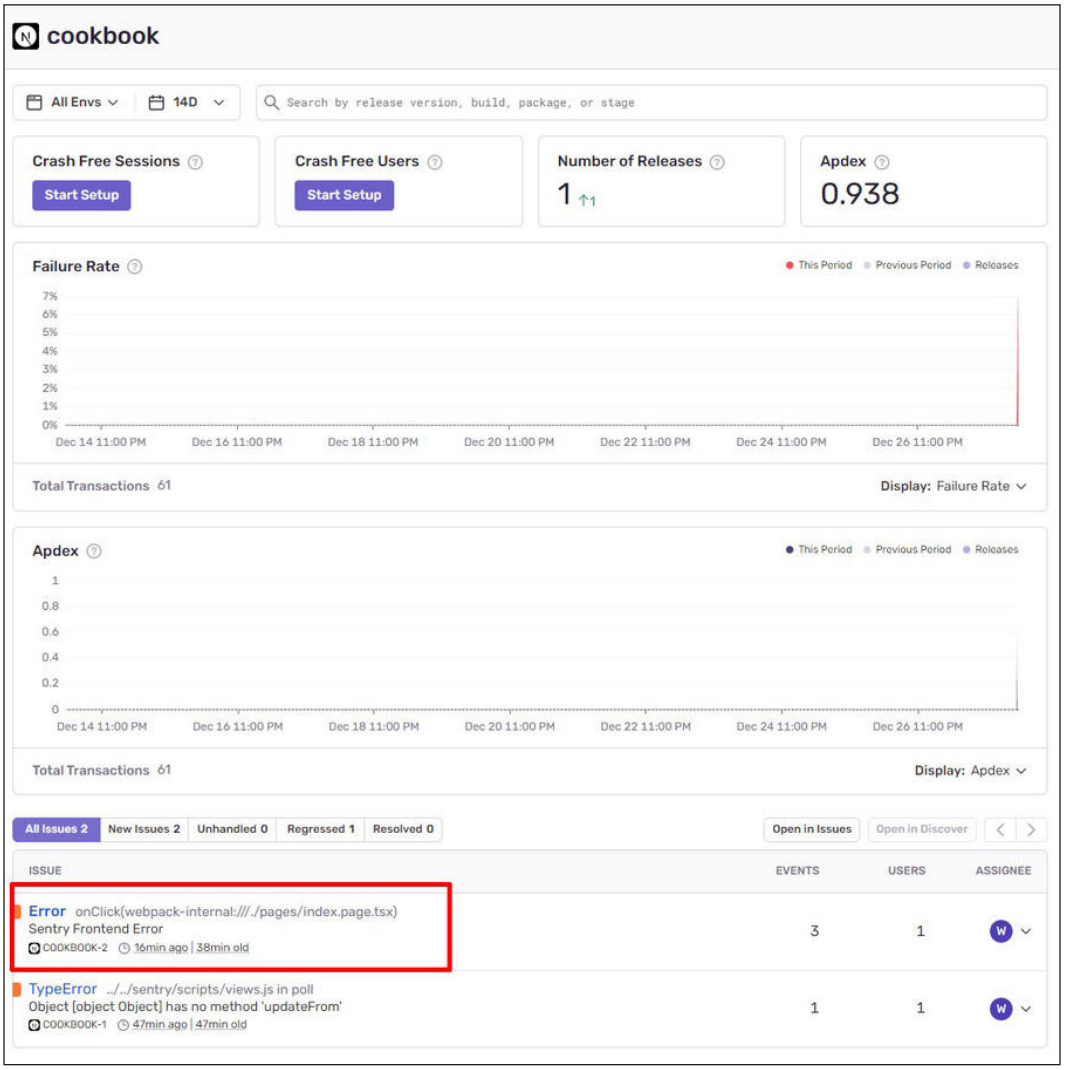


Figure 8.15: *Sentry console with a new issue in the list*

Using AWS Amplify to host our application

Before we start exploring the AWS Amplify service we will check if there is an in-house solution to host the project using Vercel possibilities. There is a reason why

we introduce AWS in this book against the Vercel delivery system and let's do some comparison. As there is a big list of the same possibilities and differences we will take only key elements that will be mainly in the choice between services. The comparison is presented in *Table 8.1:*

Feature	Vercel	AWS Amplify
Development UI and Admin UI	x	Amplify Studio
Infrastructure	Multi-Cloud	AWS
Database	x	DynamoDB
Authorization service	x	Amplify Auth

Table 8.1: *Comparison table for Vercel and AWS services*

As you can see AWS Amplify covers more features that could be critical for full-stack development, that's why we will choose and proceed with it for the delivery our application.

Understanding Amplify admin area

To get into the admin area we will need to create the AWS account first. Use **`https://aws.amazon.com/amplify/`** this link to get into the AWS amplify page. Then click the **`Create an AWS Account`** button to create the account.

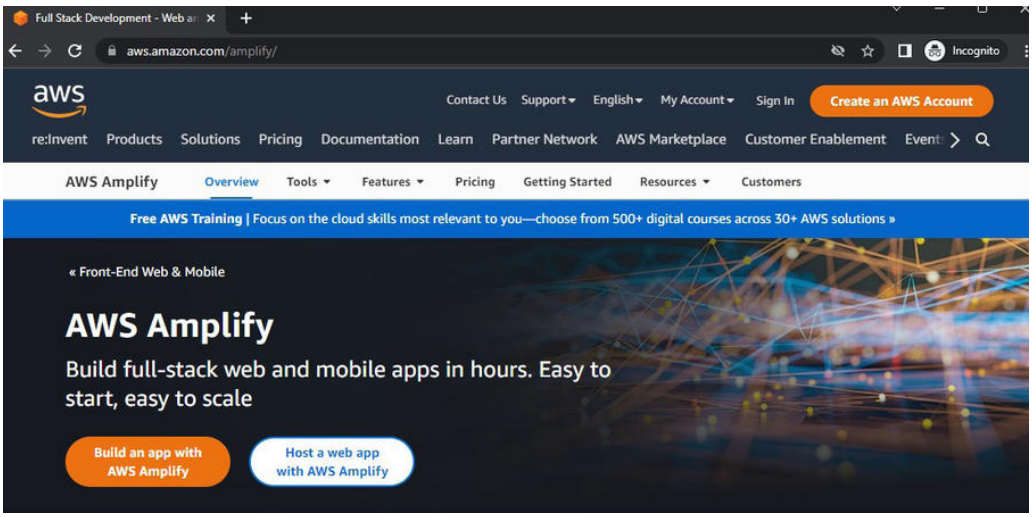


Figure 8.16: *AWS Amplify main page*

Please follow the form requirements that you will see after clicking. The form is the same as in *Figure 8.17:*

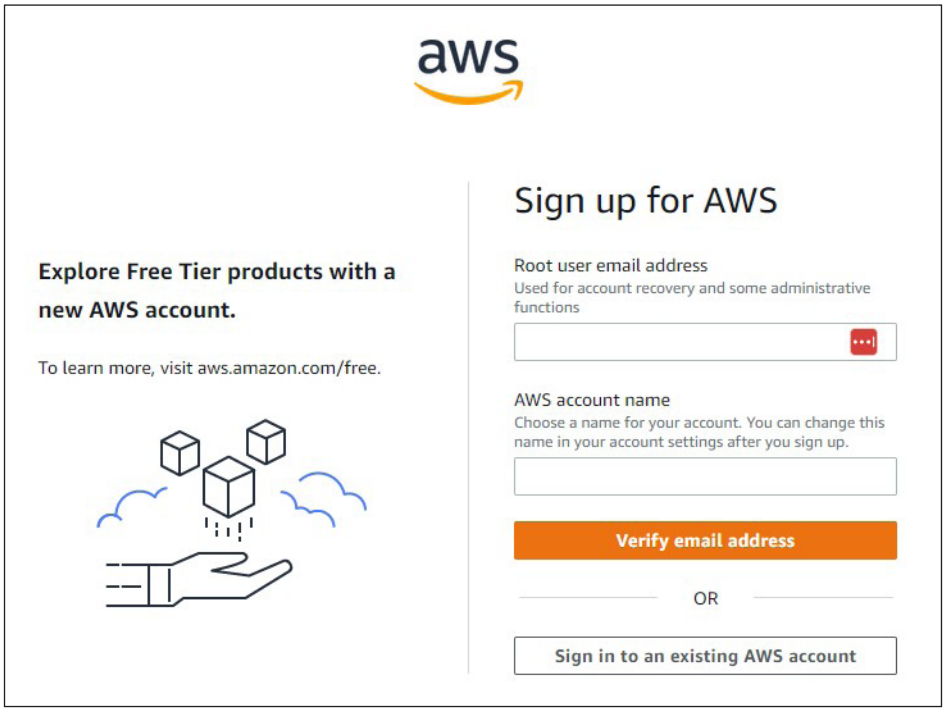


Figure 8.17: *Registration form for the AWS*

Choose the`**`Sign in`**` button if you already have an AWS account. Choose the **`Get started`**` button from *Figure 8.18* to create the new project:

Figure 8.18: *The first screen of the AWS Amplify*

Choose **`Build App`**` and you will be led to the page with application name selection. Provide the name in the form and click confirm.

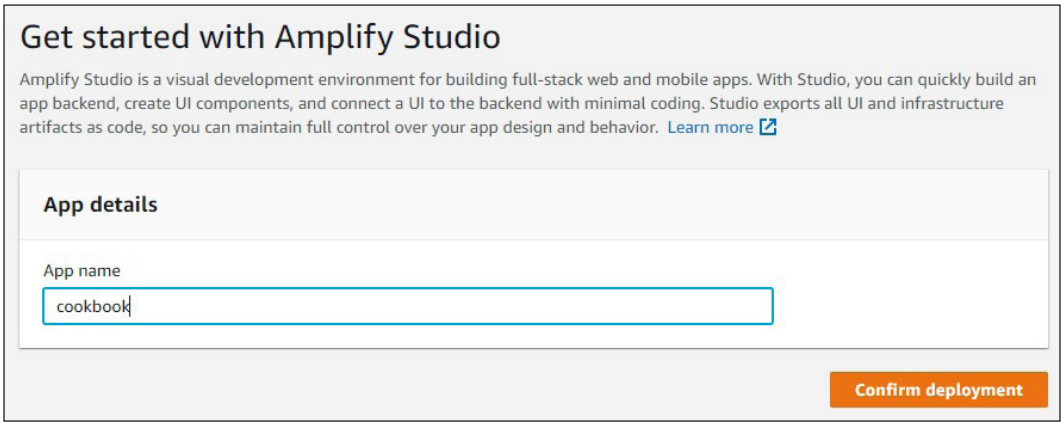


Figure 8.19: *Form to choose the project name*

After all preparation is over you will see all green stages passed as in *Figure 8.20:*

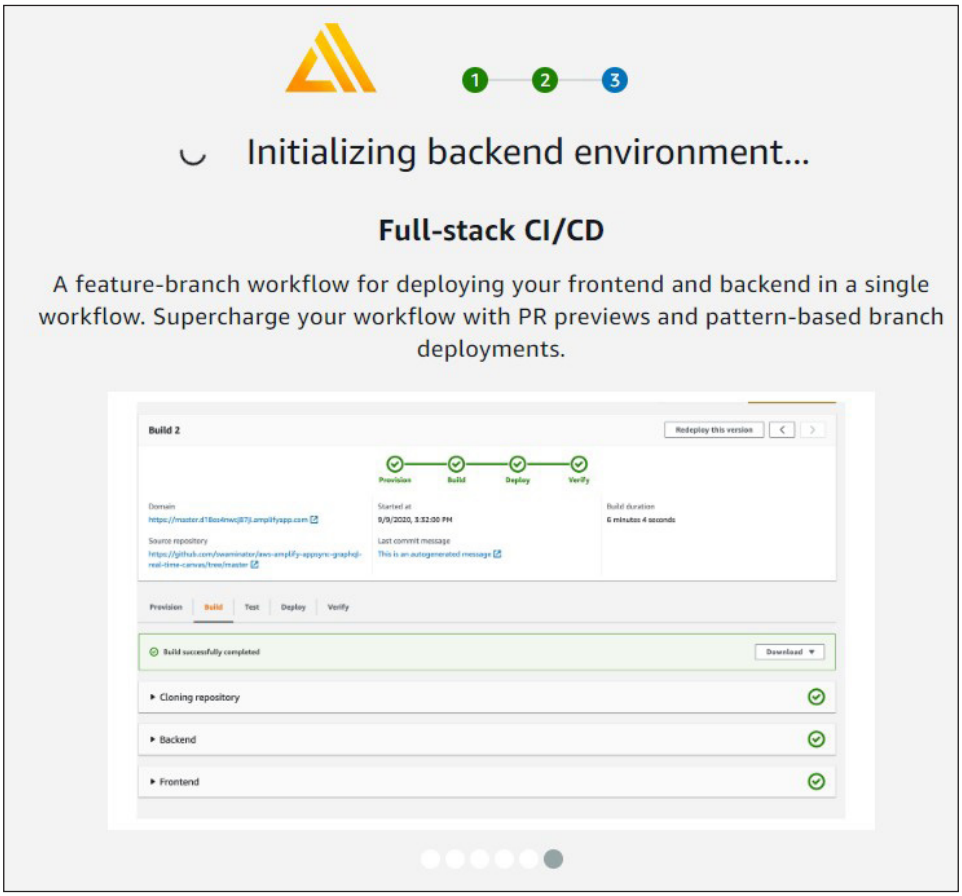


Figure 8.20: *AWS Amplify status check*

Before we connect the repository we need to figure out how to use the admin area in the amplify. Use the **`Backend environment`** tab and click on the `Launch Studio` button like in *Figure 8.21:*

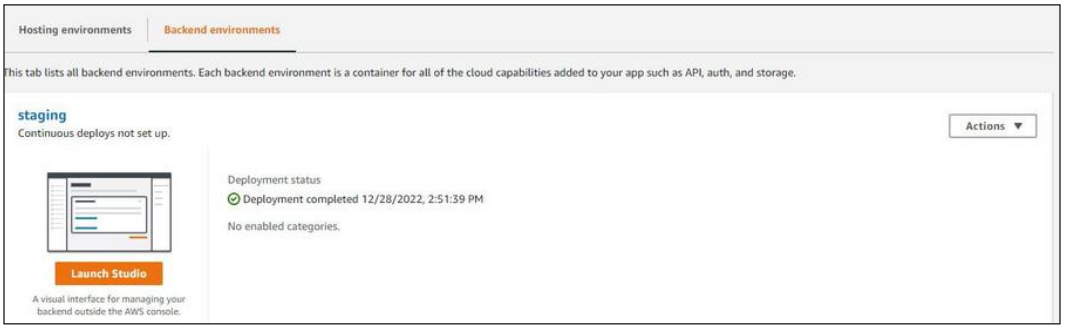


Figure 8.21: *Backend environment tab*

In the admin area you will see the screen provided in *Figure 8.22*. Let us do some research on what possibilities it has:

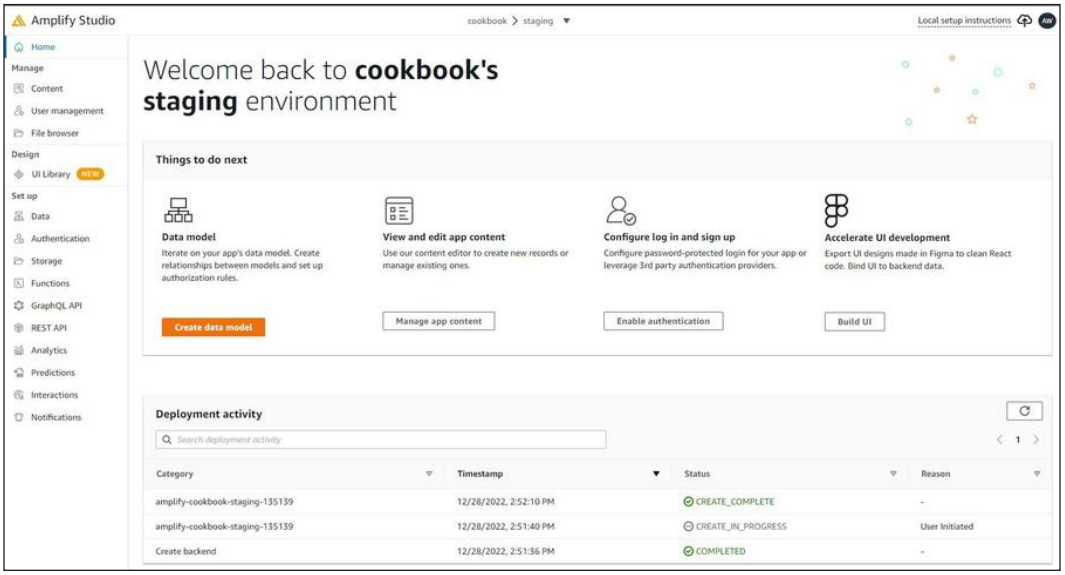


Figure 8.22: *AWS Amplify admin area*

In the left panel we will introduce the elements that will be required to cover the topic of this chapter. There are way more opportunities provided by AWS Amplify, but we can't cover all of them in the scope of this book. But you can explore the possibilities yourself:

- **Content**: this element will be required to create content for your application. This will led us to the **Data** element because we need the data structure to

store the content. All data use the same rules as any database (relations, tables.)

- **Authentication**: this element will add a possibility to authenticate users in several ways, and the users are connected to the **User management** element.
- **UI Library**: this element is required to create forms on the page to manage data in the database

Creating the data models for the application

After clicking on the Data element in the right panel we will be led to the data UI part where we can visually create the schemas. The schemas after creation will be automatically converted to GraphQL schema. Click on the `Add model` button to create a new model:

Figure 8.23: *Add model button to create a new model*

In the form that will be opened after that we need to add a name and set up the data fields that will exist in this model like in *Figure 8.24:*

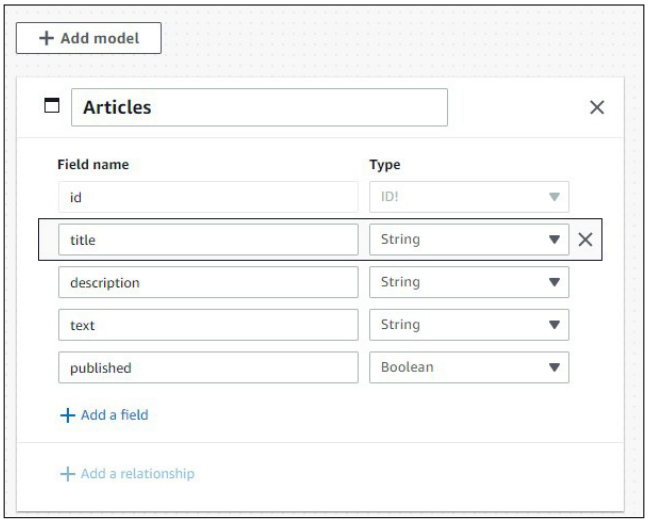


Figure 8.24: *Data model example*

Click the `Save and Deploy` button to create the data model in your project.

Creating an authentication flow with AWS

Next what we need is to set up the authentication. We will use simple authentication with email and password, but we can connect social medial login as well. In the Authentication menu element, we will not change anything. Just click the `**Deploy**` button and wait until the end of the process:

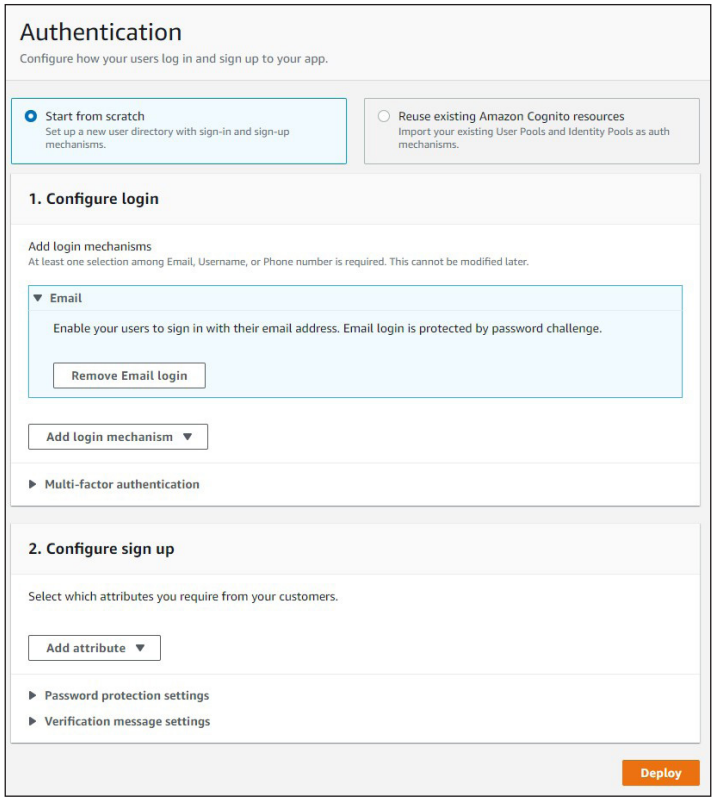


Figure 8.25: *Authentication setup in AWS Amplify*

After that we can manage users in the user's area as in *Figure 8.26:*

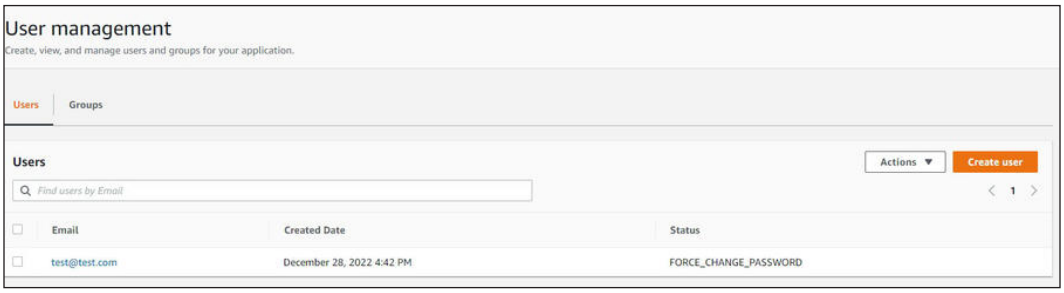


Figure 8.26: *User management area in AWS Amplify after authentication deploy*

Adding data in the admin area

In the Content area we can create new data by selecting the table in the list. For the test, we can also auto-generate content by pressing the `Auto-generate seed data` button. We will generate 10 rows for the Articles table. In *Figure 8.27*, you can see the result of the generation process:

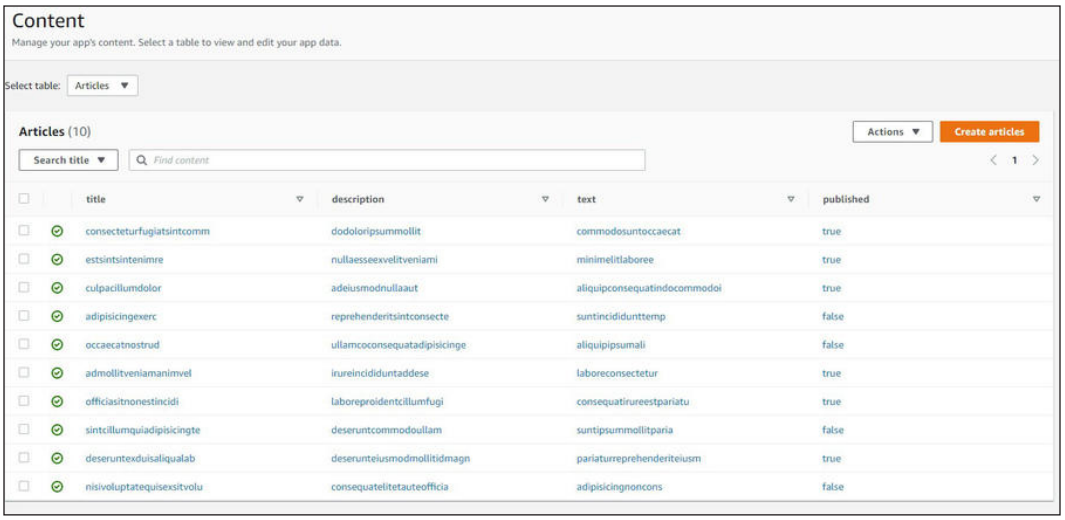


Figure 8.27: *Result of generating rows for the Articles table*

Using cloud functions for application

To start using the AWS Functions in your project that are wrapped with Amplify we need to install Amplify CLI to your computer first. Use the command from *Figure 8.28* to install Amplify:

Figure 8.28: *Command to install Amplify CLI to your computer*

Next, we need to configure the Amplify project to be connected to our local project with the command from *Figure 8.29:*

Figure 8.29: *Command to configure Amplify*

The name of your AWS zone can be collected from the interface like shown in *Figure 8.30:*

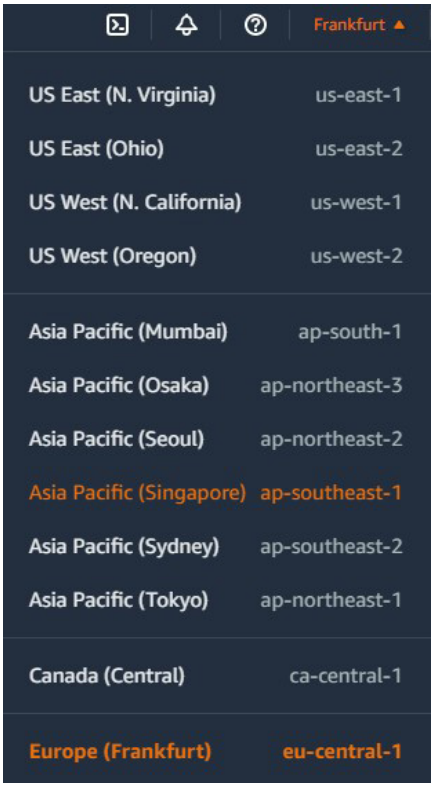


Figure 8.30: *Zone name that will be required in the configuration*

We use Europe(Frankfurt) means the zone name will be *eu-central-1*. After that, you will be led into the user check process. Follow the guide and in the form press the **`Next`** button and do not change anything. In the end, you will be on the page where the access key can be collected like in *Figure 8.31:*

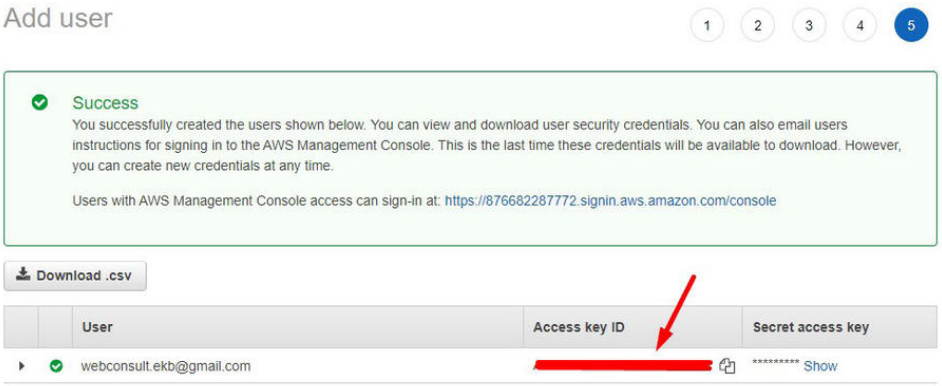


Figure 8.31: *Page with access id*

Grab this ID and provide it in your console wizard as shown in *Figure 8.32:*

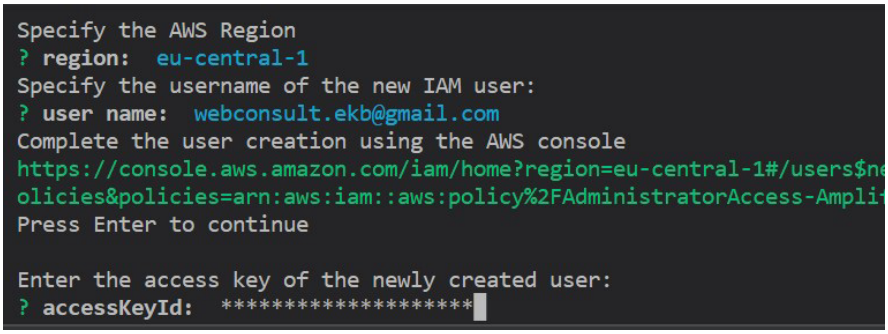


Figure 8.32: *Access Key provided in the CLI wizard*

Please also provide the secret key from *Figure 8.31* as a reply for the next question in CLI.

Next, we will initialize the amplify application with the command provided in *Figure 8.33*, please enter this command in the application root, that will be helpful in the future:

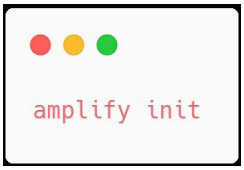


Figure 8.33: *Init the Amplify project*

In the CLI wizard you will need to enter the same information that was required previously as region, access and secret keys. You will see the success message from *Figure 8.34* when the process will be finished:

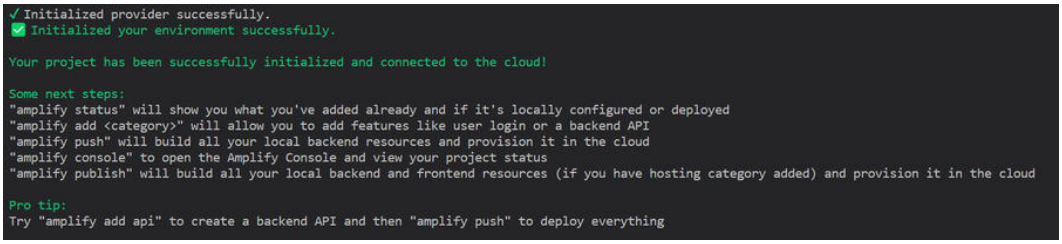


Figure 8.34: *Successful init of Amplify*

Now, we need to pull the Amplify configuration into our project. To do that grab the command in your Amplify UI from the button, that is shown in *Figure 8.35:*

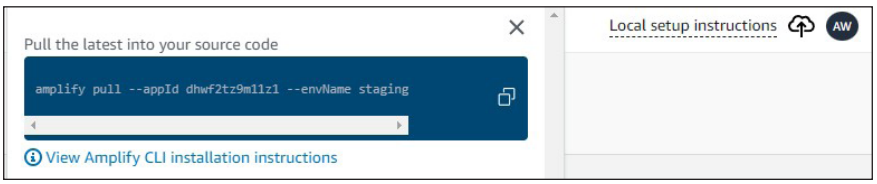


Figure 8.35: *Command to pull the project configuration*

For the question about source code please use the `**pages**` folder as the source for the NextJS app in this folder. After the configuration is finished you will see that project now have 2 more folders on the **pages** and one more folder in the root. In the source, we can find Amplify GraphQL models and the UI components to manage the data. It is the scaffolded forms that will help us to create and update data:

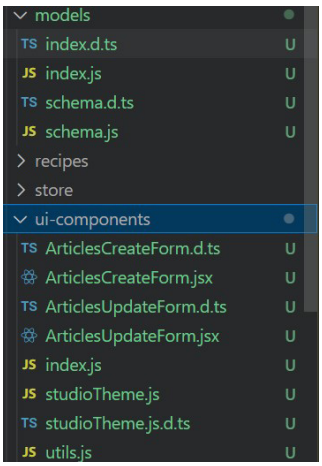


Figure 8.36: *New folders in the* ***pages*** *folder*

As the simple example of Lambda creating (the more complex is out of scope for this book, but you can learn it yourself in any AWS resource) we will create a trigger, that will update table on each data update in Articles. To do that we need to create the function by typing the command from *Figure 8.37:*

Figure 8.37: *Command to make the function*

Make selections from *Figure 8.38* to complete function creation:

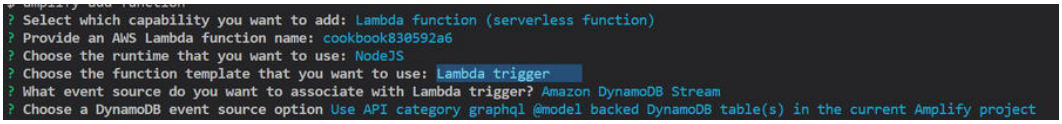


Figure 8.38*: Selections to complete the function creation*

After that, in your AWS console you can choose lambda and your function will appear in the list like in *Figure 8.39:*

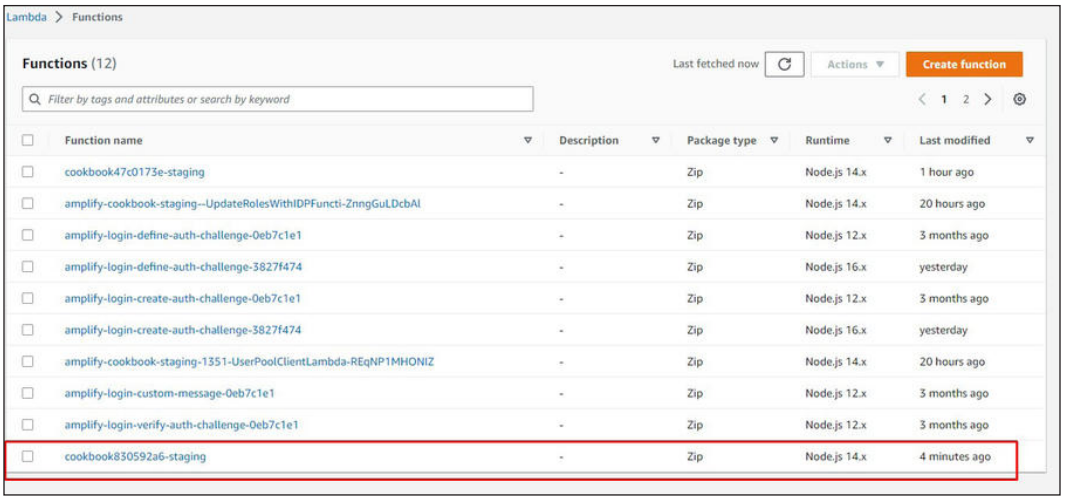


Figure 8.39: *Lambda functions list*

Click on the name and you will get inside of this function like in *Figure 8.40:*

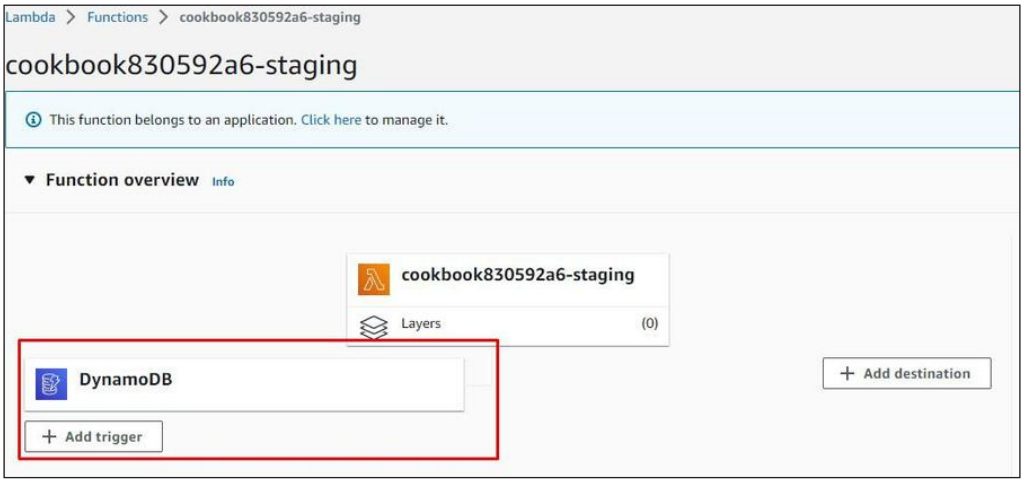


Figure 8.40: *Lambda function screen*

As you can see the trigger is already attached to the Lambda. To make changes in the other table we need to create one. Please follow the previous instructions and create a new table in the Amplify console to have it like in *Figure 8.41:*

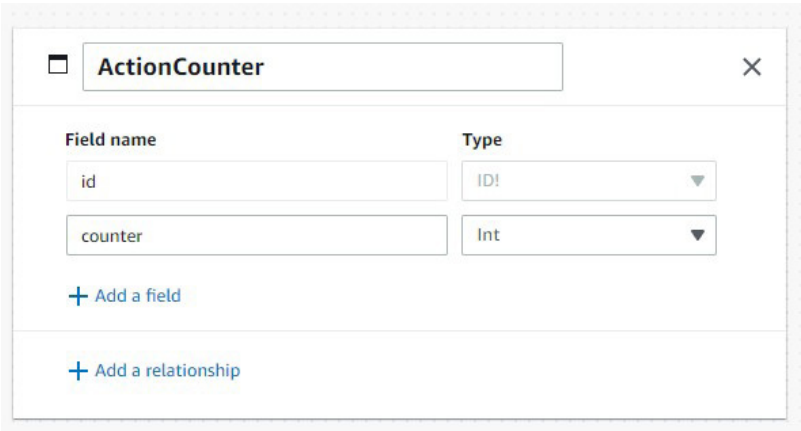


Figure 8.41: *New table to store the data*

In this table we will create one row that we will change on each article update. After that, we will create an element using the DynamoDB tab as we need to get the ID of the element that is not shown in Amplify panel. Choose DynamoDB in your AWS console and follow the DynamoDB link to get to the page from *Figure 8.42:*

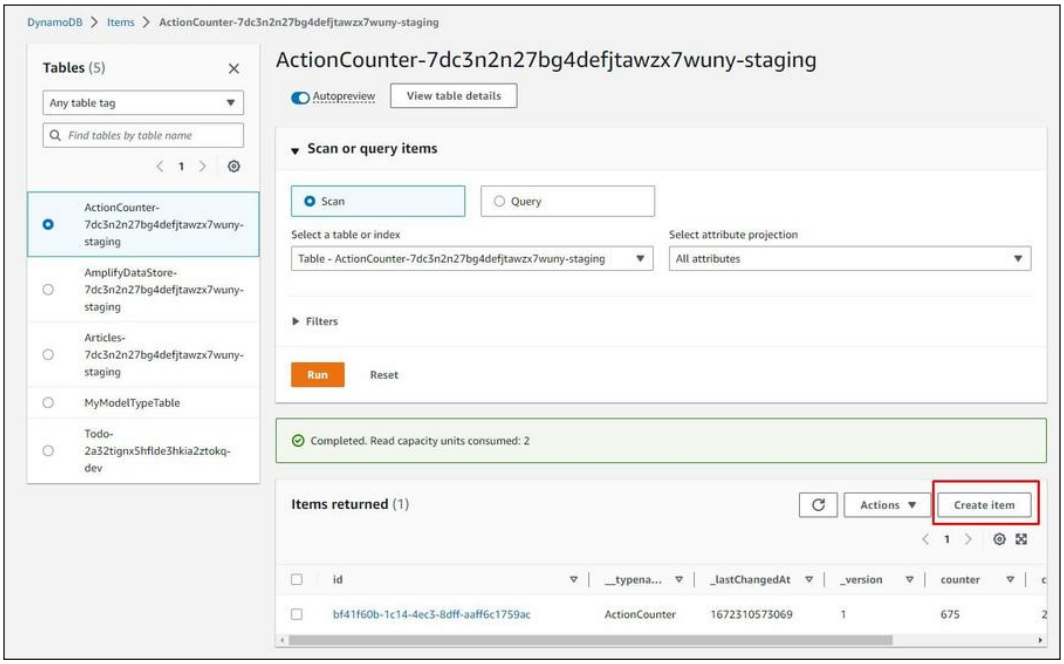


Figure 8.42: *Action counter table in AWS console*

Crate an element with any number inside. In this tab from *Figure 8.42*, we will need an element id and the table id that we will use in our Lambda function. Go back to the Lambda function panel. Here we will need to add the permission to connect to DynamoDB directly from the function. *Please note that we are doing this only for example purposes, please use an API way to get the data from the Database.*

In the **`Configuration`** | **`Permission`** tab of the Lambda function panel please click on the name of the permission to change it shown in *Figure 8.43:*

Figure 8.43: *Configuration | Permission tab of the function*

Click **`add permission and policies`** as shown in *Figure 8.44* and find the DynamoDB full access policy to add the permissions:

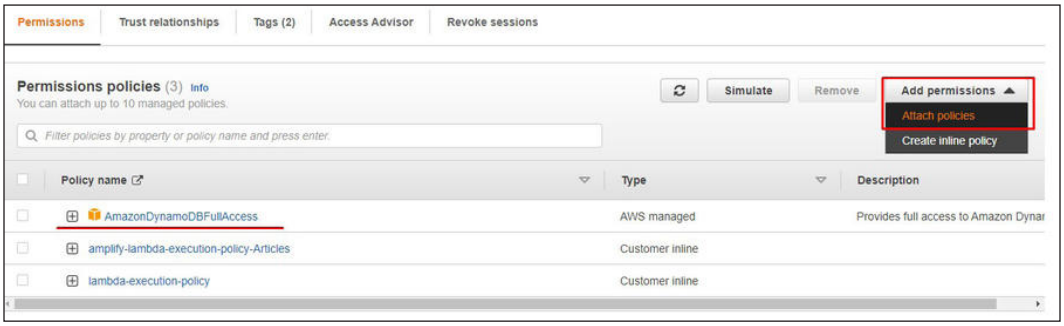


Figure 8.44: *Adding the database permissions to the function*

In the Lambda function panel please choose the Code tab as shown on *Figure 8.45*, in the code field we will place the function code:

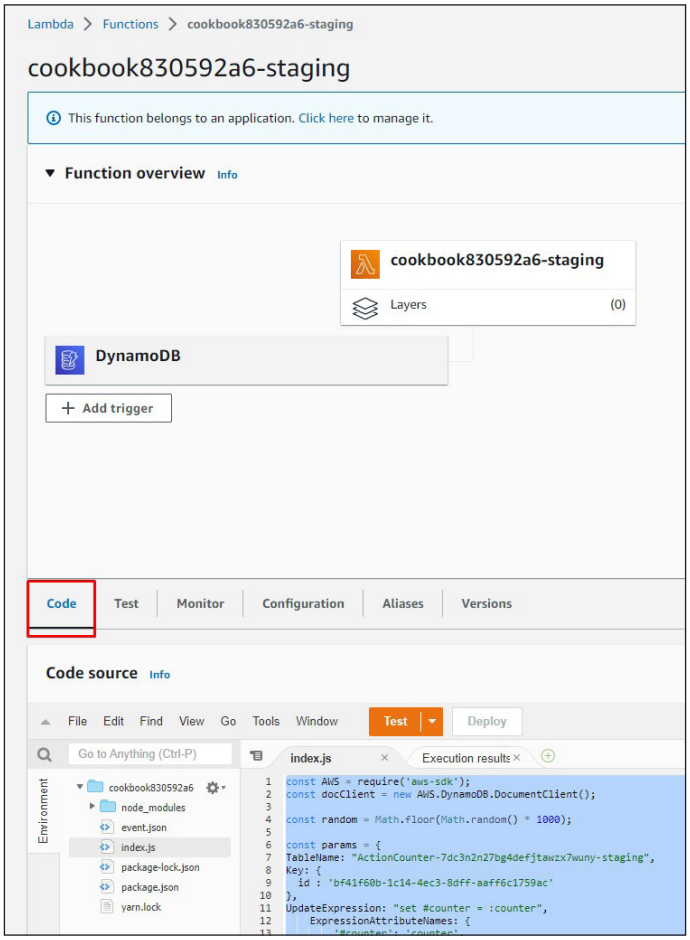


Figure 8.45: *The code tab for the lambda*

Use the code from *Figure 8.46* to your Lambda function in the code field:

```
const AWS = require('aws-sdk');
const docClient = new AWS.DynamoDB.DocumentClient();

const random = Math.floor(Math.random() * 1000);

const params = {
TableName: "ActionCounter-<your id>",
Key: {
  id : '<your item id>'
},
UpdateExpression: "set #counter = :counter",
    ExpressionAttributeNames: {
        '#counter': 'counter',
    },
    ExpressionAttributeValues:{
        ":counter": random
    },
    ReturnValues:"UPDATED_NEW"
};

async function updateItem() {
  console.log(random);
  try {
    await docClient.update(params).promise();
  } catch (err) {
    return err;
  }
}


exports.handler = async (event) => {
   try {
    const result = await updateItem();
    console.log(result);
    return { body: 'Successfully updated item!' }
  } catch (err) {
    return { error: err }
  }
};
```

Figure 8.46: *The function code*

Now you need to press on the Deploy button to deploy the function to the cloud as shown in *Figure 8.47:*

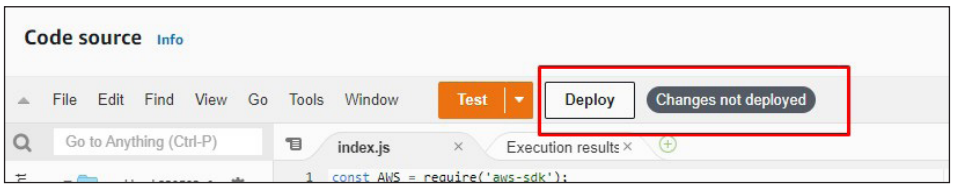


Figure 8.47: *Deploy function to the cloud*

After successfully deploying you can test the function by pressing on the Test button. To see the result open the DynamoDB console(or Amplify content) and check that number is changed. Now each time we do changes in the Article table this function will trigger changes in the other table.

Reuse cloud functions with layer functionality

For some Lambda functions we probably could require third-party npm packages to work with services or with data or, for example with dates. For the current example, we will take the MomentJS library and create a Layer with it to reuse it in any Lambda function.

The layer is a dependency container that can be reused in any Lambda function. That means that we do not need to implement dependency and install it in any function but do it once and then reuse it. We also have a limitation so in AWS we can create not more than 5 layers that can be reused.

To create the layer please create the **`nodejs`** folder in any place on your hard drive (I will put it into **`C:\temp\nodejs`**). In this folder please init the empty npm project as usual and then add the moment js package with the command from *Figure 8.48:*

Figure 8.48: *Command to install momentJS*

Your layer project should look like provided in *Figure 8.49:*

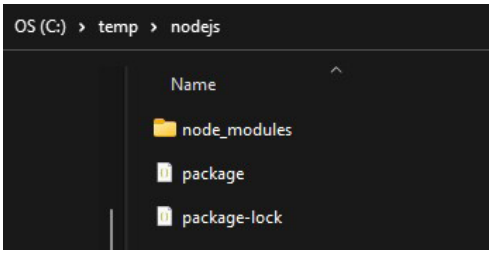


Figure 8.49: *A file structure for the Layer*

Zip the **nodejs** folder and open the AWS Lambda console. Click on the **`Layers`** menu element to open the Layers UI admin interface like in *Figure 8.50:*

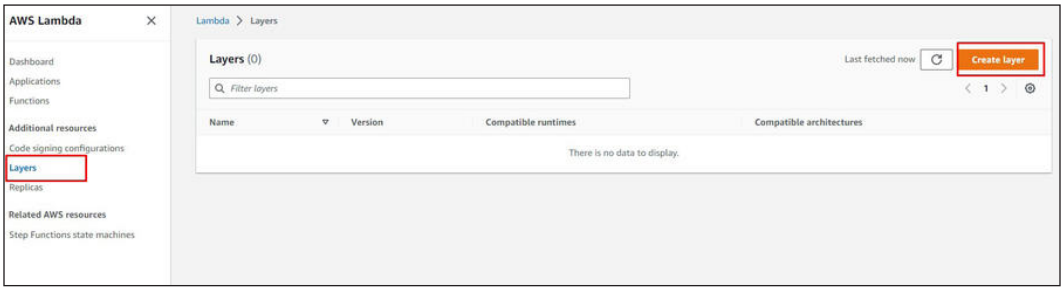


Figure 8.50: *Layers menu*

As you can see from *Figure 8.50* there is a `**Create Layer**` button. Press it to create your new layer. Fill the form as provided in *Figure 8.51* to create your first Layer:

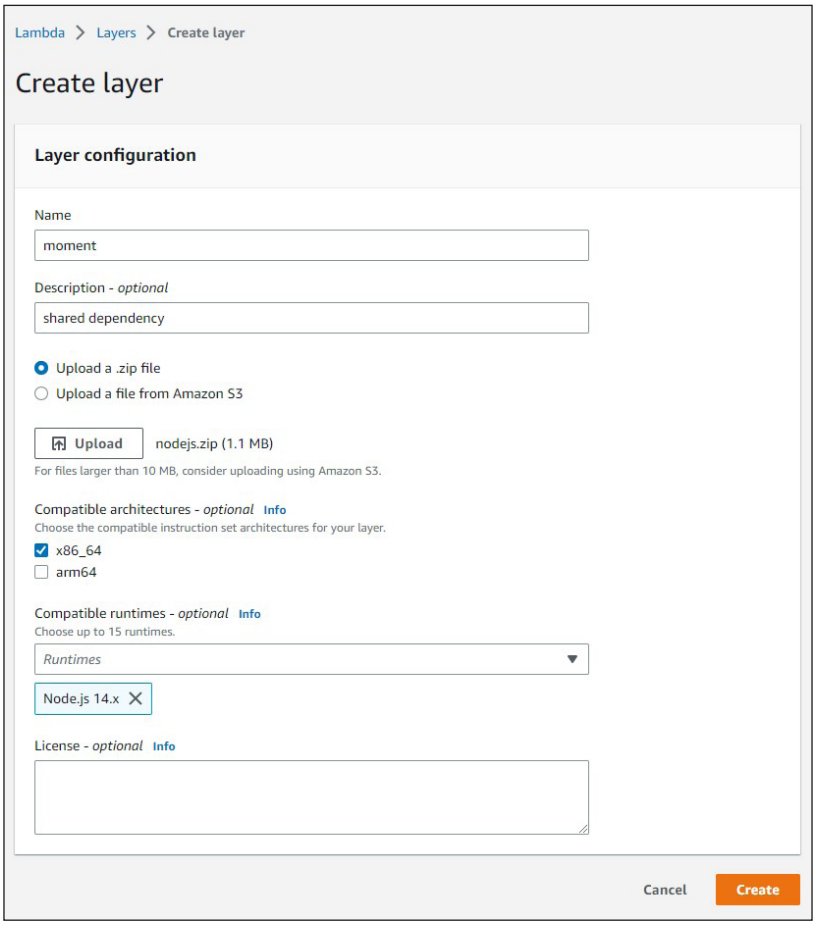


Figure 8.51: *Create layer form*

Now enter the lambda function that we created before and scroll to the Layers list selection as in *Figure 8.52:*

Figure 8.52: *Add layer group*

Click the `Add layer` button from *Figure 8.52* then choose **`Custom layers`** and find your first layer in the list like in *Figure 8.53:*

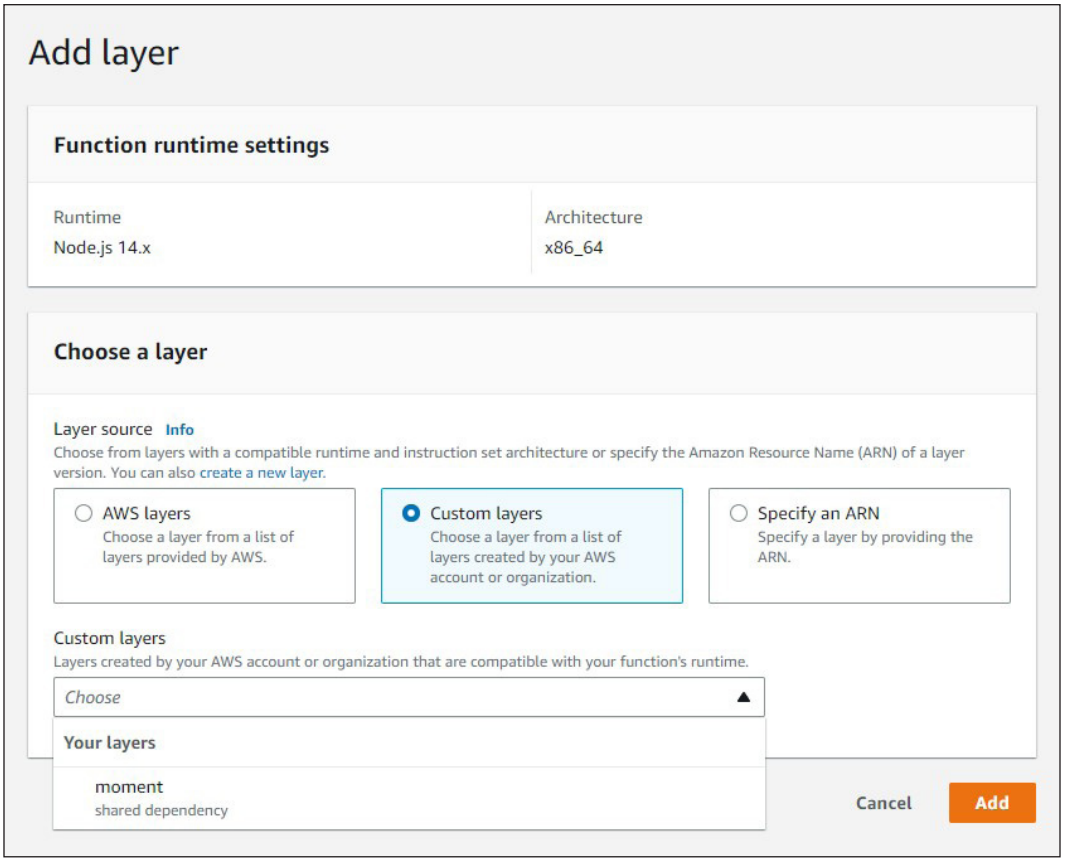


Figure 8.53: *Adding the layer to lambda function*

Press the `Add` button and switch to the `Code` tab. We will need to make some changes there.

The code changes can be observed in *Figure 8.54:*

```
// same code
const moment = require('moment');

const randomCall= moment().unix();

const params = {
TableName: "ActionCounter-<your id>",
Key: {
  id : '<your id>'
},
UpdateExpression: "set #counter = :counter",
    ExpressionAttributeNames: {
        '#counter': 'counter',
    },
    ExpressionAttributeValues:{
        ":counter": randomCall
    },
    ReturnValues:"UPDATED_NEW"
};
// same code
```

Figure 8.54: *Code updates for the Lambda function*

As you can see we import the function from the package that is not related to the current function. But we can do it because of the Layer functionality and the MomentJS is connected to it.

Finishing the backend with amplify

Let us summarize what we have now in the AWS Amplify:

- We have an articles list that we can show in our application
- We have a login system that can be implemented in the app

So to make a conclusion we need to implement this functionality in our application

To do it we need to add Amplify to our pages. The documentation for the NextJS is not so clear at the moment of the book creation so probably something will be corrected. If not then please follow this guide to easily implement Amplify into the frontend part.

First, we need to install dependencies into the project that will help us to create a connection. Use *Figure 8.55* to install the dependencies:

```
yarn add @aws-amplify/core
yarn add @aws-amplify/datastore

// OR if npm

npm install @aws-amplify/core
npm install @aws-amplify/datastore
```

Figure 8.55: *Dependencies for Amplify*

After that we need to push everything we have into the Amplify to create the configuration file. Use `**amplify push**` to push changes to the cloud. Please sure that file `**aws-exports.js**` appeared in the **pages** folder. Now, we can connect to the Amplify.

Open the **_app.page.tsx** file and put code from *Figure 8.56* before component:

```
import Amplify from '@aws-amplify/core';
import config from '../pages/aws-exports';
Amplify.configure({
  ...config, ssr: true
});
```

Figure 8.56: *Code to implement Amplify*

Now we can call Datastore to retrieve data:

```
// same code
import { DataStore } from '@aws-amplify/datastore';
import { Articles } from '../models/index';

const ListPage: NextPage = ({ data, notFound }: any) => {
  // same code
  useEffect(() => {
    const models = DataStore.query(Articles, c => c.published.eq(true));
    models.then(result => {
      console.log(result);
    });
  }, [])
  // same code
};

export default ListPage
```

Figure 8.57: *Code to retrieve Articles data*

Use *Figure 8.57* to get the data from datastore. Change the data in component to the result of the query:

```
import type { NextPage } from 'next'
import ArticleListElement from '../../ui/molecules/ArticleListElement';
import { Fragment, useEffect, useState } from 'react';
import { selectAuthState } from '../store/authSlice';
import { useAppSelector } from '../hooks';
import { DataStore } from 'aws-amplify';
import { Articles } from '../models';

const ListPage: NextPage = () => {
  const isLoggedIn = useAppSelector(selectAuthState);
  const [ data, setData ] = useState([])

  useEffect(async () => {
    const models = await DataStore.query(Articles, c => c.published.eq(true));
    setDate(result);
  });

  return (
    <section>
      <h1>Articles list</h1>
      {
        data?.map((item: any) => {
          return <Fragment key={item.id}>
                  <ArticleListElement
                    isLoggedIn={isLoggedIn}
                    article={item}
                    />
                  </Fragment>
        })
      }
    </section>
  )
}

export default ListPage
```

Figure 8.58: *Updated articles list component*

Now your list will look like in *Figure 8.59* with data from Amplify Content UI:

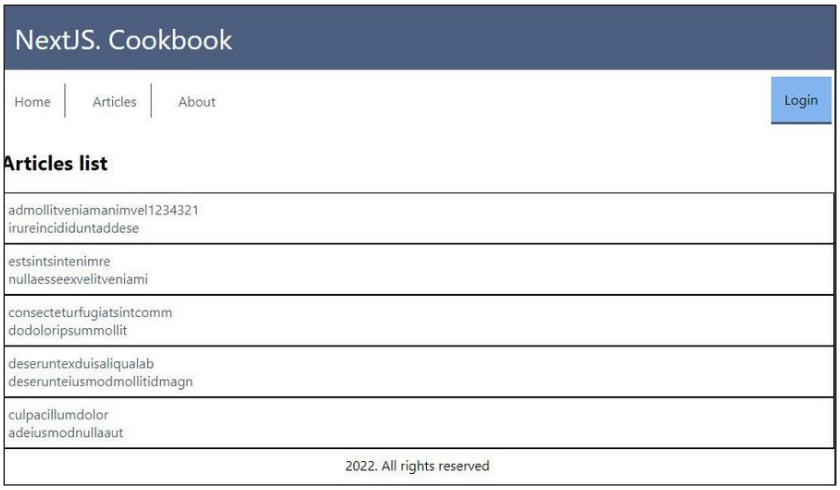


Figure 8.59: *Data list from Amplify datastore*

Next we need to update the article page as shown in *Figure 8.60:*

```
// same code as before
import { useEffect, useState } from 'react'
import { Articles } from '../models'
import { DataStore } from '@aws-amplify/datastore';

const ArticlePage: NextPage = ({ notFound }: any) => {
  // same code as before
  const [ data, setData ] = useState({});
 // same code as before

  useEffect(() => {
    const models = DataStore.query(Articles, router.query.pid);
    models.then(result => {
      setData(result);
    });
  }, [])

  return (
    <section className={styles.section}>
        // same code as before
        <div>
          <ArticleDate date={data.updatedAt} />
        </div>
      </div>
      // same code as before
  )
}

export default ArticlePage
```

Figure 8.60: *Article element change after update*

Now we can see the current article as you can observe in *Figure 8.61:*

Figure 8.61: *Current article from datastore*

Now we can update the login. To do that let us add the dependency using the command from *Figure 8.62:*

```
yarn add @aws-amplify/auth

// or if npm

npm i @aws-amplify/auth
```

Figure 8.62: *Commands to add auth dependency to the project*

As you remember we made our login system with a strategies pattern and to add the new kind of authorization we simply need to add a new strategy and use it. Please add the Strategy to the configuration file first, the name of the file is **`/pages/api/core/configuration.ts`**. Grab the code from *Figure 8.63* for the update:

```
import { LoginWithGQL, LoginWithMock, LoginWithAmplify } from "./login-strategy"
// same code as before
const LoginStrategies = {
    [LoginStrategiesNames.MOCK]: new LoginWithMock(),
    [LoginStrategiesNames.GQL]: new LoginWithGQL(),
    [LoginStrategiesNames.AMPLIFY]: new LoginWithAmplify(),
}

const loginType = LoginStrategies[LoginStrategiesNames.AMPLIFY]

export { Configuration, loginType, UserBuilderMethods }
```

Figure 8.63: *Configuration update*

Now, in the strategies file (./**login-strategy.ts**) we need to add a strategy class like in *Figure 8.64:*

```
// same code as before
import Auth from "@aws-amplify/auth";

class LoginWithAmplify implements ILoginStrategy {
    public async login(user: string, password: string) {
        let loginState = { isLoggedIn: false, token: '', userProperties: [] };
        try {
            console.log({user, password});
            const loginStateCall = await Auth.signIn(user, password);
            if (loginStateCall.Session) {
                loginState = { isLoggedIn: true, token: loginStateCall.Session, userProperties: [] };
            }
        } catch (error) {
            console.log('error signing in', error);
        }
        return loginState;
    }
}
// same code as before

export { LoginContext, LoginWithMock, LoginWithGQL, LoginWithAmplify }
```

Figure 8.64: *Login with Amplify strategy*

And that is it!! Now after login using the credentials of the user from the Amplify User panel we will be directed to the page where we can see elements only for authorized users:

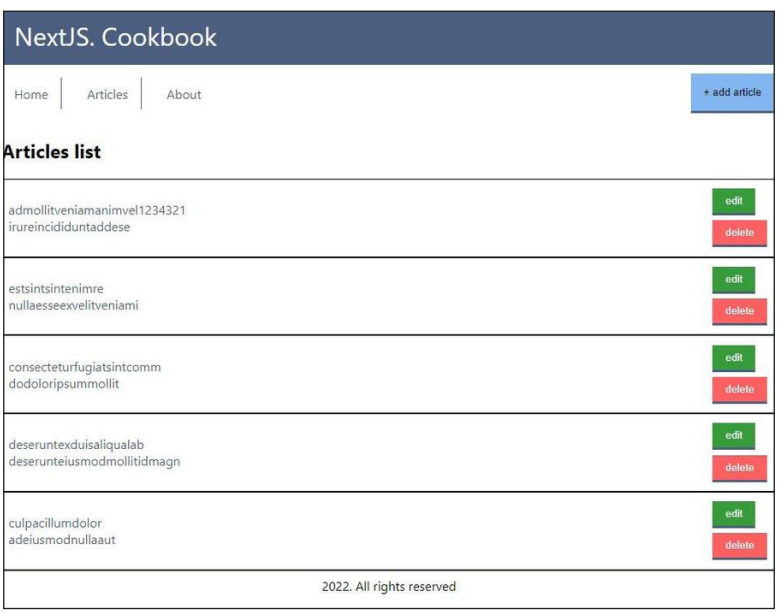


Figure 8.65: *Articles after authorization*

Host the application in the cloud and the first run

We are ready to launch our application in the cloud. To do that please put your project into any GIT services that you like (Github, Gitlab, BitBuket, and the like). We will need to connect our Amplify project with it. To connect the application with GIT please follow *Figure 8.66:*

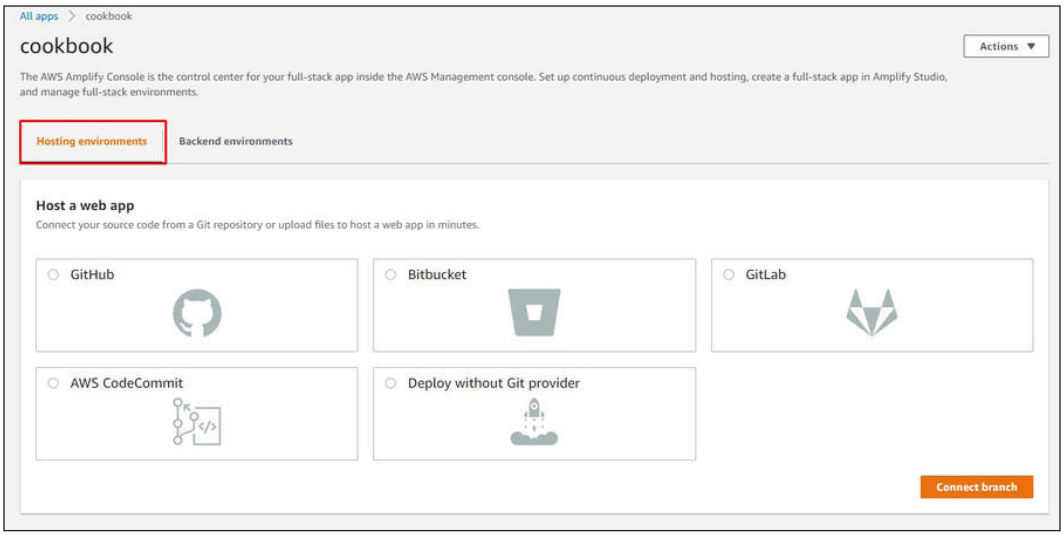


Figure 8.66: *The main page of Amplify app*

Choose the GIT provider where your project code is located.

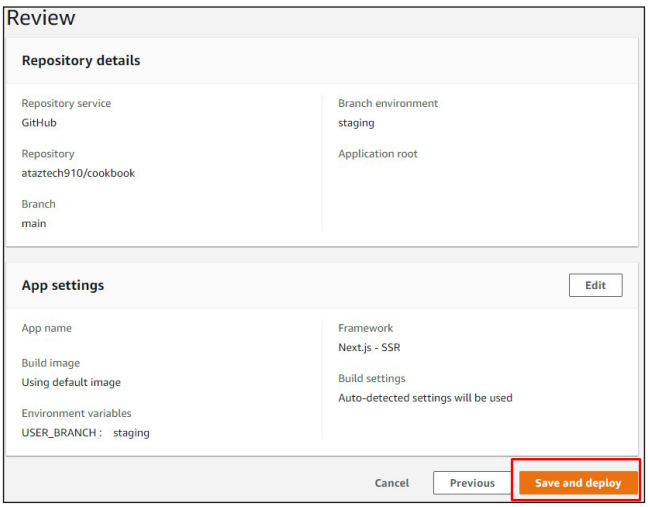


Figure 8.67: *Finishing repository connection*

After you finish the choosing repository just click `**Save and deploy**` from *Figure 8.67*. From the first Attempt, your build probably will fail. It is because of the non-documented issue with Amplify and Sentry. We need to add some environment variables to make it work in the Amplify environment. Refer to the *Figure 8.68* to solve this issue:

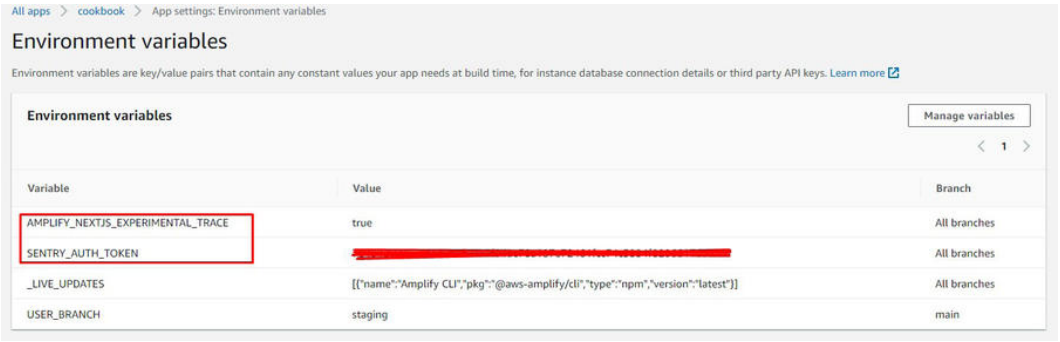


Figure 8.68: *Amplify environment variables*

You can collect your auth token from Sentry using this link:`**https://sentry.io/settings/account/api/auth-tokens/**`

In *Figure 8.69* you can see the example of success deploy (all steps should become green). And also the link to the frontend can be collected on the left side from *Figure 8.69:*

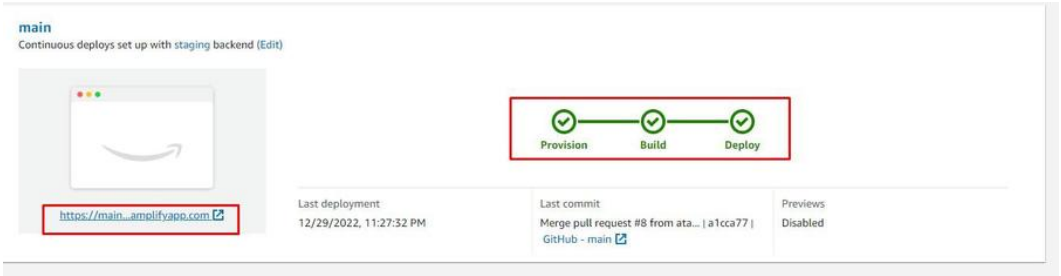


Figure 8.69: *Success deploy and link to frontend*

Click this link and you will be led to the hosted project in the staging environment like in *Figure 8.70:*

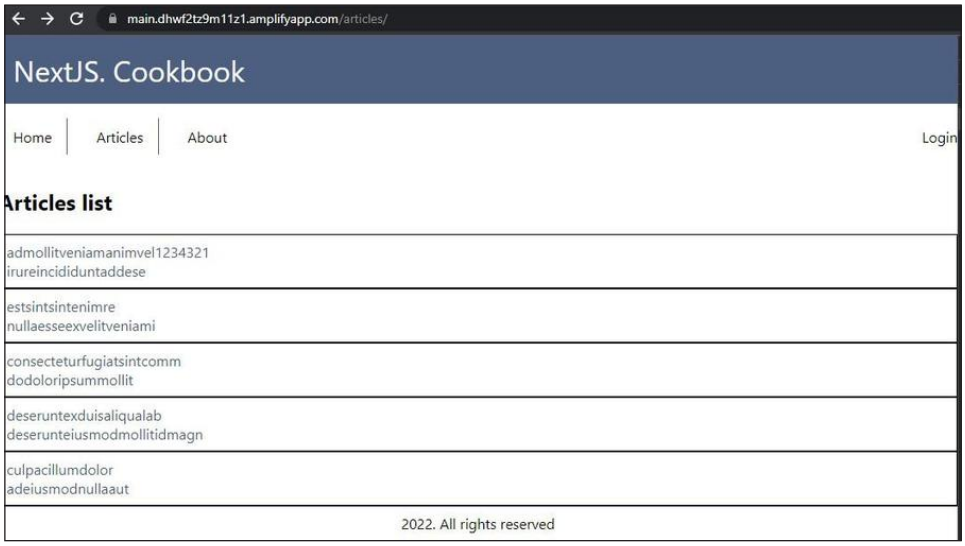


Figure 8.70: *Frontend, hosted to the Amplify*

Conclusion

In this chapter we made one of the most complicated and important parts - delivery your application to the end user. In the real life, you can choose any cloud solution that exists on the market. Any of them will have the same way of working and delivery, so if you know how to deliver to AWS, you can be sure that there will be not a big deal to switch to any other solution. Please note that Amplify today is the only all-in-one solution but you might not need the serverless solutions and backend from AWS.

In the next chapter, we will make a final touch and make our software perfect in case of optimization and SEO.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 9
Mastering optimization tools for NextJS

Introduction

It is always difficult to make the perfect even better. But we will try to improve NextJS across various parameters such as performance and optimization. Additionally, since we leverage the full power of server-side rendering, we can enhance our application for SEO purposes.

Structure

- How to get more performance from superfast NextJS
  - Using dynamic load for the client side to reduce the first load
  - How to optimize images with components
- How to bake server-side components
- Creating SEO-friendly optimization
- Conclusion

Objectives

This is the final chapter of the book and here we will try to introduce everything that will help you to make your project a little bit better. All the tools that we will introduce not have a special group so I decided to put them in the very last chapter. The reasons are:

- It is a helper but not one of the main parts of development
- There is no special group of purpose for them
- One can be sure that you are tired of so much information at the end of the book and these topics make you have fun with NextJS

How to get more performance from superfast NextJS

NextJS is a very fast framework and the reason for the performance is a not-usual list of build possibilities. We can configure the project to the strict requirements and get the exact results of performance that we need and expect. But what if there is a way to make it work faster? Not extremely faster but have a 10 or 15 percent of performance increase. There is a way to help you in some cases and let us look at them.

Using dynamic load for the client side to reduce the first load

NextJS perfectly implements one of the exciting features of modern JavaScript that allow us to defer the loading of the modules separately. This practice can divide bundles into smaller chunks that can eventually improve site performance.

Let us check what can be a drawback of performance in common projects:

- Web apps that require user interaction load a whole bunch of components even if they are not required at the current moment.
- A huge codebase creates a quite big bundle size and the compilation process is really slow because of the big number of components.

Using dynamic components led to the process called code splitting that in the end land us to reduce the main bundle size and split it into several files that load asynchronously.

For example, in our project, the top navigation bar will be loaded statically as you can see in *Figure 9.1:*

```
import styles from '../../styles/layout.module.scss'
import NavigationBar from '../../ui/molecules/NavigationBar'

export default function Layout({ children } : Partial<any>) {
    const navigation = [
      {title: 'Home', link: '/'},
      {title: 'Articles', link: '/articles'},
      {title: 'About', link: '/about'}
    ]
    return (
      <>
        <header className={styles.header}>NextJS. Cookbook</header>
        <nav>
          <NavigationBar navigation={navigation} />
        </nav>
        <main>{children}</main>
        <footer className={styles.footer}>2022. All rights reserved</footer>
      </>
    )
  }
```

Figure 9.1: *Example of the static load of the NavigationBar component*

In this case, the page will render all components before showing them on the page. To make the navigation component dynamic use the code from *Figure 9.2* to improve the component:

```
import dynamic from 'next/dynamic';
import styles from '../../styles/layout.module.scss'

export default function Layout({ children } : Partial<any>) {
    const navigation = [
      {title: 'Home', link: '/'},
      {title: 'Articles', link: '/articles'},
      {title: 'About', link: '/about'}
    ];

    const NavigationBar = dynamic(() => import('../../ui/molecules/NavigationBar'),
      { loading: () => <p>Loading navigaion...</p> });

    return (
      <>
        <header className={styles.header}>NextJS. Cookbook</header>
        <nav>
          <NavigationBar navigation={navigation}/>
        </nav>
        <main>{children}</main>
        <footer className={styles.footer}>2022. All rights reserved</footer>
      </>
    )
  }
```

Figure 9.2: *Dynamic navigation component*

As you can see in *Figure 9.2* the component is not loaded with the component but the asynchronous load on the component init. That simple optimization will lead to a various number of improvements. Be sure that in the small projects with not a bit number of components on the page you probably will not see any improvements visually, but for the bigger ones, this simple change will be a great advantage. This will solve the following:

- Conversion rate improvement. Your site will load faster and more data at one time.
- We will decrease the bounce rate which will simultaneously lead to better performance.
- We will improve the time to interaction time that will also improve the site. Next.js optimizations can reduce the time it takes for users to start interacting with your web application. A faster TTI leads to a better user experience, as users can quickly access the content and features they need.

All these improvements are not only about performance. It is also about the ranking of the site in the search engine for the robots these metrics matter and we should always keep in mind that the web application's purpose is not only for internal use and dashboards.

Let us play around with this feature to see how else we can use it. We can create the wrapper that will trigger lazy loaded in the component loaded event. That means that we will expect the promise of the component loading and then send it to the container like in *Figure 9.3:*

```
import dynamic from 'next/dynamic'
import styles from '../../styles/layout.module.scss'
import { useState } from 'react';

export default function Layout({ children } : Partial<any>) {
    const navigation = [
      {title: 'Home', link: '/'},
      {title: 'Articles', link: '/articles'},
      {title: 'About', link: '/about'}
    ]

    // Lazy loaded wrapped
    const NavigationBarWrapper = dynamic(
      () => import('../../ui/molecules/NavigationBar').then((component) => component.default)
    )

    return (
      <>
        <header className={styles.header}>NextJS. Cookbook</header>
        <nav>
        <NavigationBarWrapper navigation={navigation} />

        </nav>
        <main>{children}</main>
        <footer className={styles.footer}>2022. All rights reserved</footer>
      </>
    )
  }
```

Figure 9.3: *Lazy loaded wrapper for the component*

For this case, we can also create the whole logic that will load the required component by name depending on the route, as an example. In *Figure 9.4* you can collect the code for this example:

```
import dynamic from 'next/dynamic'
import styles from '../../styles/layout.module.scss'
import { useState } from 'react';
import { useRouter } from 'next/router';

export default function Layout({ children } : Partial<any>) {
    const navigation = [
      {title: 'Home', link: '/'},
      {title: 'Articles', link: '/articles'},
      {title: 'About', link: '/about'}
    ]

    const { route } = useRouter();
    console.log('query', route);
    const isCopy = route === '/articles'? 'Copy' : '';

    const NavigationBarWrapper = dynamic(
      () => import('../../ui/molecules/NavigationBar'+isCopy).then((component) => component.default)
    )

    return (
      <>
        <header className={styles.header}>NextJS. Cookbook</header>
        <nav>
        <NavigationBarWrapper navigation={navigation} />

        </nav>
        <main>{children}</main>
        <footer className={styles.footer}>2022. All rights reserved</footer>
      </>
    )
  }
```

Figure 9.4: *Dynamically change components by the route*

In the browser, in case you in the articles page you will see the same as in *Figure 9.5:*

Figure 9.5: *Result of dynamically loaded logic*

The number of dynamically loaded components doesn`t matter. We can load as many components as we need.

The next thing is that we can manually configure the dynamically loaded component to be client rendered and pass the SSR configuration. In *Figure 9.6* you can observe the example code of how to do that:

```
const NavigationBarWrapper = dynamic(
      () => import('../../ui/molecules/NavigationBar'+isCopy).then((component) => component.default), {
        ssr: false
      }
)
```

Figure 9.6: *Configuration for the dynamic component to be client-side rendered*

Now the navigator component will be rendered way faster than before as it is not a part of server-side rendering. This can be used for the components that are not required to be a part of the content for the SEO (shopping cart or user details for example).

The last thing that we will check in this topic is the possibility of dynamically loading third-party libraries. For example, I will take the Axios library and call a fake API with it to get the result. The example code is located in *Figure 9.7:*

```
// same code as before
export default function Layout({ children } : Partial<any>) {
    // same code as before
    let [response, setResponse] = useState([]);
    const api_url = 'https://my-json-server.typicode.com/typicode/demo/posts';

    // same code as before

    return (
      <>
        <header className={styles.header}>NextJS. Cookbook</header>
        <nav>
        Response is {JSON.stringify(response)}
        <button
        onClick={async () => {
          const axios = (await import("axios")).default;
          await axios.get(api_url).then((res) => {
            setResponse(res.data);
          });

        }}
      >
        Click me
        </button>

        // same code as before
      </>
    )
  }
```

Figure 9.7: *Dynamically load third-party library*

As you can see we are loading the Axios library only in the place where it is required to be. Be sure that this example is only the academic way of use, for the real-world application it will be strange to use it like this, but we will do it only to show you that it is a possible way of using dynamic load. *Figure 9.8* is the result of clicking on the button:

Figure 9.8: *Result of clicking on the button where the library loaded by clicking on it*

How to optimize images with components

The basic problems that can be met with images for your site can be grouped in the list like this:

- The format of the image is chosen incorrectly. In some cases, PNG is way bigger than the same image in JPEG format. On the modern web, it is better to use the WebP format as it is the most optimized format for the web. You can find numerous converters on the internet that could help you to translate images to WebP.
- The wrong size of the images led to an increased loading time for the page. For example, we do not need 4k images for mobile users with a maximum of 1440p screens. We can detect the device and provide the required one. There are a lot of services that will help you to resize your image to have it for each required screen size.
- The wrong compression of images could be also an issue, so before translating the image to the WebP and creating the bundle for several resolutions use any compression service on your image to reduce size. A lot of images are having information that is not visible to the average human eye, so it is just there and can be easily removed with compression.

In NextJS there is a special component, that can solve a lot of problems, so we highly recommend using it.

Let us try to use it, but first, we will get the image in WebP format. After that, we place the file in the public folder. Use code from *Figure 9.9* to insert the image:

```
// Same code as before
import Image from 'next/image'
import imageExample from '../../public/file_example.webp'

export default function Layout({ children } : Partial<any>) {
   // Same code as before
    return (
      <>
      // Same code as before
        <Image
                src={imageExample}
                alt='user random picture'
        />
       // Same code as before
      </>
    )
  }
// Same code as before
```

Figure 9.9: *Insertion of the image using the Image component*

This component automatically will create several parameters like width, height, and **`dataBlurUrl`**. This is very important for the CLS metric from the previous chapter. If you use remotely located images (for example: from S3 or any other file store) please always put width and height for the component to avoid CLS degradation. Let us look at the possible properties of the component that can be useful:

- **`src`**: here you can provide a statically imported file or string with a URL to the remote storage.
- **`layout`**: string property to configure responsibility for the image. Please observe possible values for this property.
- **`intrinsic`**: it is the default value that renders enough space to use the original size of the image.
- **`fixed`**: fills the parent's size. Please make sure that the parent element is having `position: relative` property.

- **`responsive`**: reacts to parent element width. Make sure your parent container is having `display: block property.
- **`loader`**: this parameter generates a loader element before the image is loaded, but as a parameter, it can take configuration variables from the NextJS. Check the example code in *Figure 9.10*.
- **`placeholder`**: this parameter will generate a way of loading the image visually. This parameter has 2 options:
  - **`empty`**: nothing will be visually shown
  - **`blur`**: the image will be blurred until the load is over. You can use **`blurDataUrl`** param to show any other image you want
- **`priority`**: this parameter will disable lazy loading and put these images in the loading queue higher than others.
- **`quality`**: this parameter is to manage image quality. The range of it is between 1 and 100. Changing this parameter also affects the image file size, so you can reduce the quality for the images where you need it to be small but not in high-resolution.
- **`sizes`**: this parameter's purpose is to set sizes like minimal or maximal width. This parameter replicates a standard sizes param that is used in HTML IMG tag.
- **`loading`**: this param is configuring a loading type for the image.
  - **`lazy`**: default type that loads the image asynchronous
  - **`eager`**: if this is selected then the image will be loaded synchronously and hurt performance
  - **`objectFit`**: this property replicates the CSS object-fit behavior so you can select fill cover or contain the same as you would do that with CSS.
  - **`objectPosition`**: also replicates the CSS object-position property.
- **`onLoadingComplete:`** this is the callback function property. That means that after the image is loaded we can call some function (for example if more than 10 images are loaded then we can take more data from API)

As you can see using the **`Image`** component from NextJS is more performant and configurable. Also, you will need fewer components to work and manage images as most functionality is already inside.

```
import Image from 'next/image'

const customLoader = ({ src, width, quality }) => {
  return `<url-to-image-server>/${src}?
w=${width}&q=${quality}`
}

const ImageWithLoader = (props) => {
  return (
    <Image
        src="someImage.webp"
        width={300}
        height={300}
        alt="example image"
        quality={80}
        loader={customLoader}
    />
  )
}
```

Figure 9.10: *Image loader example code*

How to bake server-side components

This is an experimental feature, so it is required React 18 which will come with Next 13. At the moment of this book creation, the 12th version of NextJS is stable so I would highly recommend using this one for a while. But let us look into the future and check what is covered under the newest React and NextJS.

Before we start we need to turn on the feature in the NextJS config using the configuration object from *Figure 9.11:*

```
experimental: {
    appDir: true,
}
```

Figure 9.11: *Configuration for the NextJS*

Just add this in the configuration in the **`next.config.js`** file. We will also need the latest version of NextJS so please remove the current version and add ^13.0.3. Next what you will need to do is to rename the **pages** folder to the **app** and make huge reconfigurations of the folders.

If you do not want to break the current project you can also use the example from Vercel that is located at this link `**https://github.com/vercel/next-react-server-components**`. It is much better than renaming everything in the current project. This is what I will do in the next steps to not to break the current project as it has way much setup that could be broken during reconfiguration.

So the server-side rendered components are the components that are literally stored and rendered on the server.

What this feature can give us in the future:

- Direct access to the database that makes fetching faster (can be used for non-critical components that can have direct access to DB).
- The Server side components are not included in the bundle so you cannot load them on the page during the load stage. That also means that we could not use any client-side interactivity and hooks like useState or useEffect.

To create the server-side component we need to create a new folder in the file structure that is called the **server** and put the page component there as in *Figure 9.12:*

Figure 9.12: *New file structure for the server-side component*

Grab the code from *Figure 9.13* as the component code:

```
export default function Server() {
    console.log('Server page rendering: this should only be printed on the server');
    return (
        <div>
            <h1>Server Page</h1>
        </div>
    );
}
```

Figure 9.13: *Server-side component code*

The magic trick is that when you will enter the `/server` link in the console you will not see any text from the component code. This log will be only on the server side.

```
Server page rendering: this should only be printed on the server
Server page rendering: this should only be printed on the server
```

Figure 9.14: *Log of server console that will not appear on the client side*

Next what we can do is mix server-rendered components with client components. To do that please create the **mix** folder and put the **page** file inside (if you use Vercel example then put page.js there). Refer to the *Figure 9.15* for the code:

```
import Client from './client';
import Server from './server';

export default function MixMatchPage() {
    console.log('MixPage rendering');
    return (
        <div>
            <h1>Server Page</h1>
            <div className="box">
                <Client message="A message from
server">            <Server />
                </Client>
            </div>
        </div>
    );
}
```

Figure 9.15: *Code for the mixed component*

As you can see we will need two more components. The first step - will be the client component and the second one is the server.

For the client please create the **`client.js`** file and use *Figure 9.16* and copy the code from it:

```
'use client';

// This import here just to show that compiler will not
// comlain on useEffect
import { useEffect } from 'react';

export default function Client({
    message,
    children,
    }) {
    console.log('Client component rendering');

    return (
        <div>
            <h2>Client Child</h2>
            <p>Message from parent: {message}</p>
            <div className="box-red">{children}</div>
        </div>
    );
}
```

Figure 9.16: *The client component code*

For the server - please create a **`server.js`** file in the **`mix`** folder and use code from *Figure 9.17:*

```
export default function Server() {
    console.log('Server component rendering');
    return (
        <div>
            <h3>Server</h3>
            <p>Server content</p>
        </div>
    );
}
```

Figure 9.17: *A server component*

As you see in the Client component we used the `**`use client`**;` property to indicate the compiler to use this component as client-rendered. After entering `**`/mix`**` page you will see the result from *Figure 9.18* with mixed components:

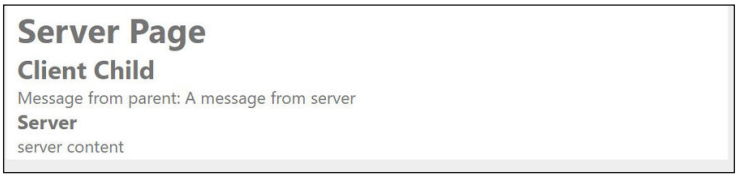


Figure 9.18: *Mixed components*

Please note that if you try to put a server component into the client component - the server component will automatically degenerate into the client component. So be careful with it and find the correct places for your components.

Interesting fact, using the server-rendered components led to the thinking that NextJS is trying to reproduce the same way of using web apps as we did it 10 years ago, by using server pages technologies like Java or PHP. Will see how far it will come.

Creating SEO-friendly optimization

To make here some recommendations let us describe what is SEO and why is it important. **SEO** is **Search Engine Optimisation**. Basically, everything in this chapter (except experimental features) stands to improve the site optimization that will directly affect SEO.

As SEO is a very huge topic that deserves a separate book, we will be going through only the several parts that will help you to improve your site.

First what we will improve is a head part of the page that will contain the title and description for the page. First, we need to create this part to use. Please observe *Figure 9.19* to collect the code with changes:

```
// Same code as before
import Head from 'next/head'

export default function Layout({ children } : Partial<any>) {
    // Same code as before
    return (
      <>
        <Head>
            <title>Create Next App PWA</title>
            <meta name="description" content="Generated by create next app" />
        </Head>
        <// Same code as before
      </>
    )
  }
```

Figure 9.19: *Updates for the Layout component with head*

After that you will see that title and description exist on each page. The next step is to add an Open graph element to the header. Open graph elements are special elements that will help your page to become a rich object in the social graph. This functionality will help your page to be shown correctly in Google (to have the correct title, image, and description), and also if someone will share your page it will be shown correctly with the correct data and image.

- **`og:title`**: The title of your object which should appear within the graph, for example "The Rock".
- **`og:type`**: The type of your object, for example "video.movie". Depending on the type you specify, other properties which may also be required.
- **`og:image`**: An image URL that should represent your object within the graph.
- **`og: URL`**: The canonical URL of your object that will be used as its permanent ID in the graph, for example, **"https://example.com/somepage-123/".**

Refer to the *Figure 9.20* to update your header with the following code:

```
<Head>
    <title>Create Next App PWA</title>
    <meta name="description" content="Generated by create next app" />
    <meta property="og:title" content="Create Next App PWA" />
    <meta property="og:description" content="Generated by create next app" />
    <meta property="og:URL" content="https://nextjs-cookbook.site/" />
    <meta property="og:type" content="website" />
</Head>
```

Figure 9.20: *Open graph tags for the head element*

The problem now is that we will have the same information for each page. To solve it we need to make these tags generate dynamically.

Update the component with code from *Figure 9.21* to solve it:

```
import styles from '../../styles/layout.module.scss'
import NavigationBar from '../../ui/molecules/NavigationBar'
import Head from 'next/head'
import { useRouter } from 'next/router';

export default function Layout({ children } : Partial<any>) {
    const navigation = [
      {title: 'Home', link: '/', meta: {
        title: 'This is main page',
        description: 'This is main description'
      }},
      {title: 'Articles', link: '/articles', meta: {
        title: 'This is articles page',
        description: 'This is articles description'
      }},
      {title: 'About', link: '/about', meta: {
        title: 'This is about page',
        description: 'This is about description'
      }}
    ]

    const router = useRouter();
    const meta = navigation.find(element=> element.link === router.pathname)
    console.log('meta', meta);

    return (
      <>
        <Head>
            <title>
              {meta?.meta.title}
            </title>
            <meta name="description" content={meta?.meta.description} />
            <meta property="og:title" content={meta?.meta.title} />
            <meta property="og:description" content={meta?.meta.description} />
            <meta property="og:URL" content="https://nextjs-cookbook.site/" />
            <meta property="og:type" content="website" />
        </Head>
        <header className={styles.header}>NextJS. Cookbook</header>
        <nav>
          <NavigationBar navigation={navigation} />
        </nav>
        <main>{children}</main>
        <footer className={styles.footer}>2022. All rights reserved</footer>
      </>
    )
  }
```

Figure 9.21: *Updated component to generate meta dynamically*

Now if you check the page you will see that data is generated by the request as in *Figure 9.22:*

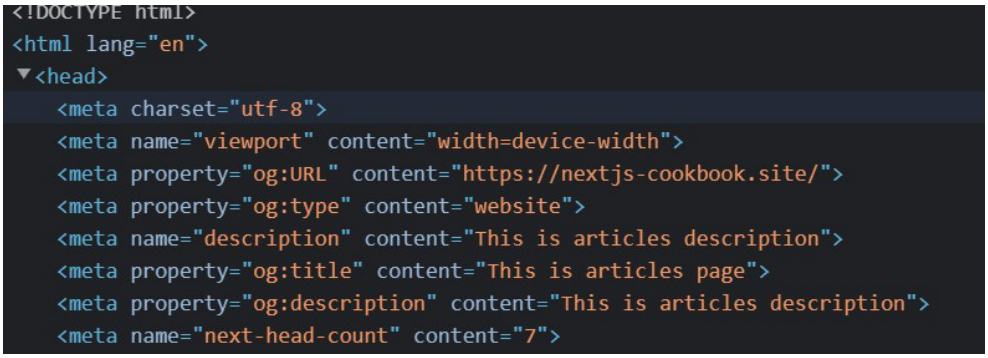


***Figure 9.22**: Generated metadata in head tag*

Conclusion

In this chapter, we managed to make last preparations and fixes that allow your project on NextJS to be one of the most performant and SEO friendly. We covered a lot of topics that could require more deep investigation.. There is no way to perfect software without everyday improvements, do not hesitate to do that!.

As it is the last chapter of the book I want to say thank you if you read this book from start to end. I hope everything that you met in this book will inspire you to create the most interesting and perfect software using the NextJS framework.

Thank you one more time, and see you in the other books!

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

R

S

T

U

W

Next.js Cookbook

DESCRIPTION

Next.js is a powerful and flexible framework for building server-side rendered React applications. In this book, you will learn how to develop a full-stack ERP application from scratch to production using Next.js.

The book will begin by covering the basics of Next.js, including setting up the environment and creating your first app. You will then learn how to use design patterns to optimize the application development process. Next, the book will help you get familiar with Next.js's server-side rendering capabilities by providing the knowledge and skills needed to leverage this powerful feature to improve the performance and user experience of your ERP application. You will also learn to manage the complex application state using Redux, as well as how to implement internal pages and create a CRUD system for user data. In addition, the book will help you perform end-to-end testing using Cypress and Playwright, and deploy your application to production using AWS Amplify. Lastly, you will learn how to optimize your web application for search engines, enabling better visibility and higher traffic to your website.

By the end of the book, you will be able to develop high-quality web applications using Next.js.

KEY FEATURES

- Learn how to develop an Enterprise Resource Planning (ERP) application using Next.js.
- Learn how to use design patterns in Next.js effectively.
- Learn how to implement server-side rendering for improved performance.

WHAT YOU WILL LEARN

- Learn how to use Redux for state management in your Next.js applications.
- Learn how to create pages in Next.js with ease.
- Learn how to write end-to-end tests for your app.
- Learn how to deploy your application to production using AWS Amplify.
- Learn how to use optimization tools to improve the SEO of your application.

WHO THIS BOOK IS FOR

This book is for anyone interested in learning how to develop full-stack web applications using Next.js. It is also for technical architects, project managers, and other professionals who want to gain a deeper understanding of the technologies and best practices involved in building modern web applications.

BPB PUBLICATIONS
www.bpbonline.com

ISBN 978-93-5551-845-3