



Egidijus Svedas

Building an application using Next.js

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

23 September 2024

Author:

Egidijus Svedas

Title:

Building an application using Next.js

Number of Pages:

24 pages + 1 appendices

Date: 23 September 2024

Degree: Bachelor of Engineering
Degree Programme: Information Technology
Professional Major: Software Engineering
Supervisors: Janne Salonen, Principal Lecturer

Since its first release in 2016, the framework Next.js, which was created by Vercel, has provided React-based applications with static website generation and SSR (server-side rendering).

It is one of the recommended frameworks mentioned in the official React documentation.

React is a JavaScript library, although many people interchangeably call it a framework.

It is used to build web applications. It has some noticeable problems such as the users having disabled JavaScript, search engine optimization, extended page load times, and security issues.

Next.js solves some of these issues by rendering the side on the server, before sending it to the browser. It is, to this day, the most popular framework for React.

Keywords: React, JavaScript, Next.js

Contents

List of Abbreviations

| | | |
|-------|---|----|
| 1 | Introduction | 1 |
| 2 | Background | 1 |
| 2.1 | Evolution of Web Development Frameworks | 2 |
| 2.2 | Introduction to Next.js | 2 |
| 3 | Study and Usage of Next.js | 3 |
| 3.1 | Overview of Next.js | 3 |
| 3.2 | Features of Next.js | 3 |
| 3.2.1 | Server-Side Rendering (SSR) | 3 |
| 3.2.2 | Static Site Generation (SSG) | 3 |
| 3.2.3 | API Routes | 4 |
| 3.2.4 | Dynamic Routing | 4 |
| 3.3 | Performance of Next.js | 4 |
| 3.4 | Usage of Next.js in the Application | 4 |
| 3.4.1 | Justification for Choosing Next.js | 4 |
| 3.4.2 | Implementing Core Features with Next.js | 5 |
| 4 | Integrating Key Technologies with Next.js | 5 |
| 4.1 | Using React with Next.js | 5 |
| 4.1.1 | Component-Based Architecture | 6 |
| 4.1.2 | Hooks and State Management | 7 |
| 4.2 | Using Node.js with Next.js | 7 |
| 4.2.1 | Server-Side Capabilities | 8 |
| 4.2.2 | API Development | 8 |
| 4.3 | Using Express.js with Next.js | 9 |
| 4.3.1 | Middleware Integration | 9 |
| 4.3.2 | Custom Server Setup | 10 |
| 4.4 | Using MongoDB with Next.js | 11 |
| 4.4.1 | Database Integration | 11 |
| 4.4.2 | Data Management and Storage | 12 |

| | | |
|-------|--|----|
| 5 | Deployment and Maintenance | 13 |
| 5.1 | Preparing the Application for Deployment | 13 |
| 5.1.1 | Environment Configuration | 14 |
| 5.1.2 | Build Optimization | 14 |
| 5.2 | Deployment Strategies | 15 |
| 5.2.1 | Vercel | 15 |
| 5.2.2 | Custom Server Deployment | 15 |
| 5.3 | Continuous Integration & Continuous Deployment (CI/CD) | 17 |
| 5.4 | Application Maintenance and Monitoring | 18 |
| 5.4.1 | Error Tracking | 18 |
| 5.4.2 | Performance Monitoring | 19 |
| 5.4.3 | Keeping Dependencies Up-to-Date | 20 |
| 5.5 | Conclusion | 21 |
| 6 | Summary | 21 |
| 6.1 | Overview of Next.js | 22 |
| 6.2 | Key Components and Technologies | 22 |
| 6.3 | Development Process | 22 |
| 6.3.1 | Setting Up the Project | 22 |
| 6.3.2 | Building the Application | 22 |
| 6.3.3 | Deployment | 23 |
| 6.4 | Challenges and Solutions | 23 |
| 6.5 | Advantages of Next.js | 23 |
| 6.6 | Future Work and Recommendations | 23 |
| 6.7 | Conclusion | 24 |
| | References | 25 |

List of Abbreviations

JS: JavaScript

API: Application Programming Interface

CI/CD: Continuous Integration / Continuous Deployment

CSR: Client-Side Rendering

HTTP: Hypertext Transfer Protocol. An application protocol for distributed, collaborative hypermedia information systems used for data communication in the World Wide Web.

DOM: Document Object Model

JSON: JavaScript Object Notation

JSX: JavaScript XML

MVC: Model-View-Controller

SSR: Server-Side Rendering

SSG: Static Site Generation

REST: Representational State Transfer. An architectural style. It is used in the creation of web services where the requesting systems can access and manipulate web resources through a uniform and predefined set of stateless operations.

HTML: Hypertext Markup Language. The main markup language for creating web pages and all sorts of web applications.

CSS: Cascading Style Sheets. A style language, used in combination with HTML, to describe the way documents are represented on a web page.

1 Introduction

In the constantly evolving world of web development, choosing the right framework is crucial for building scalable, efficient, and maintainable applications. Next.js is a popular React-based framework. It is a solution for developers who want to use the benefits of SSR (server-side rendering) and SSG (static site generation). The goal of the thesis is to explore the process of building an app using the Next.js framework while displaying its main features, strengths, weaknesses, and practical implementation in a real-life app.

The goal, thus, is to provide a comprehensive and detailed guide for developers, or anyone really, looking to implement Next.js in their projects. We will do in-depth research of Next.js' main features, performance advantages, and integration with other technologies. The thesis will demonstrate how Next.js can be effectively used to build modern web apps.

2 Background

2.1 Evolution of Web Development Frameworks

For the last few decades, web development has been constantly developing. The first websites were built using static HTML. It served its purpose at the time but provided limited user interaction and required manual updates for content changes. The demand for dynamic content and more interactive websites grew. Server-side scripting languages like PHP, ASP, and others emerged, which enabled developers to create richer applications (Flanagan, 2020).

The introduction of client-side JavaScript frameworks, such as Backbone, Angular, React, and Vue.js, started a new era in web development. These frameworks allowed developers to start building single-page applications (SPAs). They have greatly improved user experiences by dynamically updating the content without full-page reloads like before (Wilson & Harbison, 2018).

However, SPAs had other challenges - slower load times and bad search engine optimization (SEO) (Clark, 2021).

To solve these problems, a framework like Next.js was developed. It combined the advantages of both server-side and client-side rendering. Next.js, built on top of React, gives us a great way to implement SSR & SSG, enhance UI performance, SEO, and the overall user experience (Vercel, 2024).

2.2 Introduction to Next.js

An open-source framework developed by Vercel, Next.js enables developers to create React applications with built-in support for SSR and SSG. It simplifies setting up a server-rendered React app, giving us a developer-friendly environment with features like automatic code splitting, dynamic routing, and static exporting (MongoDB, 2024).

One of the most popular benefits of Next.js is the ability for us to pre-render pages at build time or on each request using SSG or SSR accordingly. This greatly improves SEO and results in much faster load times. Next.js can also support API routes, thus allowing developers to build backend functionality in the same project. This lets us improve and speed up the whole development process of the app (Smith & Johnson, 2019).

To summarise, this current chapter provided us with an overview of the overall evolution of web development frameworks. It also introduced Next.js as a powerful and modern tool for building heavy UI web apps in combination with React. In the next chapters, we will have the opportunity to further investigate the other features and usages of the Next.js framework.

3 Study and Usage of Next.js

3.1 Overview of Next.js

Next.js is a basically React.js framework. With it's help, we can make the process of building high-performance and SEO-friendly apps more simple and easy. It gives us many great features.

For example:

SSR, SSG, dynamic routing, and API routes.

Next.js is designed to improve developer productivity. It does so by providing sensible defaults and practical tools for building both static & dynamic apps.

The absolute most popular Next.js feature is the approach to rendering.

Developers can choose to either render pages on the server during the request time (SSR), at build time (SSG), or even by using both methods.

It all comes down to what the application needs at that point. This kind of robust flexibility allows for optimized performance, SEO, and scalability (Clark, 2021).

3.2 Features of Next.js

3.2.1 Server-Side Rendering (SSR)

The SSR technique lets us generate HTML on the server using for each request. This greatly improves the initial load times of the app and also improves it's SEO. Search engines can then easily index the pre-rendered outputted HTML content. SSR can be implemented by exporting an `async getServerSideProps` function from a page component. This function fetches data at request time and passes it as props to the component, ensuring that the page is fully rendered on the server before being sent to the client (Smith, 2024).

3.2.2 Static Site Generation (SSG)

This technique allows devs to pre-render pages at build time. This results in faster load times since the HTML is already generated and served as static files (Wilson & Harbison, 2018). Next.js makes it easy to implement SSG by using the `getStaticProps` and `getStaticPaths` functions. `getStaticProps` fetches data at build time and passes it as props to the component, while `getStaticPaths` generates dynamic routes based on the data. SSG is very useful for content-heavy websites where we don't need to change the content frequently.

3.2.3 API Routes

API routes enable developers to create backend endpoints directly within the Next.js project. API routes are stored in the `pages/api` directory and can handle HTTP requests, making it possible to build full-stack applications with Next.js. This feature simplifies the development process by allowing both frontend and backend code to coexist in the same repository.

3.2.4 Dynamic Routing

Dynamic routing in Next.js allows developers to create routes based on dynamic data. With file-based routing, Next.js automatically maps files in the `pages` directory to corresponding routes. Dynamic routes can be defined using brackets in the file name (e.g., `[id].js`), which are then matched to URL parameters. This feature is very useful for app with user-generated content or any scenario where the content structure is not fixed (MongoDB, 2024).

3.3 Performance of Next.js

Next.js was designed with performance in mind. The hybrid rendering capabilities ensure that applications are optimized for both speed and SEO. Pre-rendering pages either on the server or at build time, Next.js reduces the time it takes for the users to see the content, resulting in a better user

experience. Additionally, Next.js automatically code-splits pages, loading only the necessary JS for each page, thus improving performance (Vercel, 2024).

3.4 Usage of Next.js in the Application

3.4.1 Justification for Choosing Next.js

Next.js was chosen for this application due to its ability to provide both SSR and SSG, improving performance and SEO (Smith & Johnson, 2019). Its developer-friendly features like automatic code splitting, dynamic routing, and API routes, improve the development process and enhance productivity. Furthermore, it has great documentation and active community support, which make Next.js a reliable choice for building modern web app.

3.4.2 Implementing Core Features with Next.js

To demonstrate the practical application of Next.js, we will implement core features such as dynamic routing, API integration, and pre-rendering. The chapter will walk us through the process of creating a Next.js project app. In it, we will create dynamic pages, and also build API routes. Readers will have a good understanding of how to use Next.js to build efficient, robust, and scalable web apps by the end of the chapter.

4 Integrating Key Technologies with Next.js

We will now explore how the Next.js framework can be integrated with various other technologies to build a full-stack app. We'll cover the integration with the most popular technologies such as: React, Node, ExpressJs & MongoDB. In each part, we will thoroughly look into the practical implementations and the advantages that these technologies bring, when used together with Next.js.

4.1. Using React with Next.js

Most importantly, Next.js is built on top of React. It is created to be used together with React. We will now explore how the component-based architecture & hooks of React could be used within a Next.js app.

4.1.1. Component-Based Architecture

The component-based architecture of React promotes the development of reusable and modular UI components (Wilson & Harbison, 2018). With Next.js, each page is a separate React component. This allows developers to build complex UI by composing many smaller self-contained components.

Example: Creating a Reusable Component

```
// components/Button.js
import React from 'react';

const Button = ({ onClick, children }) => {
  return (
    <button onClick={onClick} className="btn">
      {children}
    </button>
  );
};

export default Button;

// pages/index.js
import React from 'react';
import Button from '../components/Button';

const HomePage = () => {
  const handleClick = () => {
    alert('Button clicked!');
  };

  return (
    <div>
      <h1>Welcome to the Next.js app</h1>
      <Button onClick={handleClick}>Click Me</Button>
    </div>
  );
};

export default HomePage;
```

In the above example, a Button component is created. It is reusable and can be used inside the HomePage component. This kind of modular approach makes the code easier to manage and subsequently, maintain (Clark, 2021).

4.1.2. Hooks and State Management

One of the most popular React hooks, `useState` & `useEffect`, give us a simple way to manage component state & side effects in functional components. Next.js, of course, fully supports React hooks, thus enabling developers to handle state and lifecycle methods in a very efficient and practical manner.

Example: Using Hooks in a Next.js Component

```
// pages/counter.js
import React, { useState, useEffect } from 'react';

const CounterPage = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log(`Count: ${count}`);
  }, [count]);

  return (
    <div>
      <h1>Counter</h1>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>Increment by one</button>
    </div>
  );
};

export default CounterPage;
```

We can see in this example that the “CounterPage” component uses the “`useState`” hook to manage the count state and the `useEffect` hook to log the count value whenever it changes. The code thus demonstrates how we can use hooks to handle the state & side effects in a Next.js app.

We also have libraries like Redux or Context API, that can be used together with Next.js for global state management (Wilson & Harbison, 2018). The libraries gives us a unified way to manage state across the whole app. It makes it easier to share state between components in big and complex apps.

4.2. Using Node.js with Next.js

Node can be used with Next.js for server-side operations. It would enable backend possibilities within the framework, making the application not only the frontend but full-stack. So in next section, we will look into how we can do that.

4.2.1. Server-Side Capabilities

Node.js is used for server-side rendering in Next.js. It gives us the ability to execute JS code on the server before sending the HTML to the client (Smith, 2024). This improves the whole performance of the app and the SEO by delivering fully rendered pages.

Example: Implementing Server-Side Rendering

```
// pages/ssr.js
import React from 'react';

const SSRPage = ({ data }) => {
  return (
    <p>
      <h2>SSR. Server-Side Rendering</h2>
      <p>Data fetched from server: {data}</p>
    </p>
  );
};

export async function getServerSideProps() {
  const result = await fetch('https://api.example.com/data');
  const data = await result.json();

  return {
    props: { data }
  };
}

export default SSRPage;
```

In this example we can see that the SSRPage component uses `getServerSideProps` to fetch data from an API on a server. After that, the fetched data is passed as props to the component, which is rendered on the server before being sent to the client (Flanagan, 2020).

4.2.2. API Development

Using Next.js, we can create API routes within the `pages/api` directory. These routes are just Node.js functions, which can handle HTTP requests. This gives us the possibility to develop the backend in the same project (Express.js, 2024).

Example: Creating an API Route

```
// pages/api/hiya.js
export function handler(request, result) {
  result.status(200).json({ message: 'Hiya everyone!' });
}
```

Here we can see how an API route is created. It basically just handles GET requests and responds with a simple JSON message. This demonstrates how easy it is to set up an API route with Next.js. We can now develop backend functionality within the same project.

4.3. Using Express.js with Next.js

Next.js already comes with a built-in server, but integrating Express.js can sometimes provide more flexibility and control over the backend server logic. So in the next section, we shall look at how Express.js middleware & custom server setups can enhance a Next.js app.

4.3.1. Middleware Integration

Express.js has middleware functions. They can be used to handle tasks such as auth, logging, and request parsing. If we create a custom server with Express.js, the developers can use middleware to extend the functionality of their Next.js apps.

Example: Integrating Middleware

```
// server.js
const express = require('express');
const next = require('next');

const dev = process.env.NODE_ENV !== 'PROD';
const app = next({ dev });
const handle = app.getRequestHandler();

app.prepare().then(() => {
  const server = express();

  server.use((req, res, next) => {
    console.log('Request has been received: ${req.method} ${req.url}');
    next();
  });

  server.all('*', (req, res) => {
```

```

        return handle(req, res);
    });

    server.listen(5000, (error) => {
        if (error) throw error;
        console.log('> Ready. http://localhost:5000');
    });
});

```

So here we have created a custom server using Express.js. Middleware is then used to output the incoming requests using the `console.log()` function. So we can see how Express.js can improve the server-side functionality of a Next.js app (Wilson & Harbison, 2018).

4.3.2. Custom Server Setup

To setup our own custom server with Express.js using Next.js we need to create a `express` instance and use it to handle requests. This allows devs to define custom routes, handle server-side logic & integrate with other Node.js modules appropriately.

Example: Custom Routes with Express.js

```

// server.js
const express = require('express');
const next = require('next');

const dev = process.env.NODE_ENV !== 'PROD';
const app = next({ dev });
const handle = app.getRequestHandler();

app.prepare().then(() => {
    const server = express();

    server.get('/some-custom-route', (request, result) => {
        return app.render(request, result, '/custom', request.query);
    });

    server.all('*', (request, result) => {
        return handle(request, result);
    });

    server.listen(5000, (error) => {
        if (err) throw error;
        console.log('> Ready. http://localhost:5000');
    });
});

```

Here we can see that a custom route `/some-custom-route` is defined with Express.js. This route renders a specific page, demonstrating how custom server setups can provide additional routing possibilities in a Next.js app (Clark, 2021).

4.4. Using MongoDB with Next.js

We can integrate MongoDB, a NoSQL DB, with Next.js. It would be used to manage and store the app data in an efficient way. This next section will explore how to connect MongoDB to a Next.js app.

4.4.1. Database Integration

To connect MongoDB to a Next.js app we will use a library called Mongoose. We could also use the MongoDB Node.js driver (Smith & Johnson, 2019). This library/driver provides us with tools for schema definitions, data validation, and query building. It also simplifies interaction with the DB.

Example: Connecting to MongoDB using Mongoose

```
// lib/mongodbconnection.js
import mongoose from "mongoose";

const MONGODB_CON = process.env.MONGODB_URI;

if (!MONGODB_CON) {
  throw new Error('Please define a MONGODB_CON environment var');
}

let cache = global.mongoose;

if (!cache) {
  cache = global.mongoose = { conn: null, promise: null };
}

async function connectToDatabase() {
  if (cache.conn) {
    return cache.conn;
  }

  if (!cache.promise) {
    const opts = {
      useUnifiedTopology: true,
      bufferCommands: false,
      useNewUrlParser: true,
      bufferMaxEntries: 0,
```



```

        useFindAndModify: false,
        useCreateIndex: true,
    });

    cache.promise = mongoose.connect(MONGODB_URI,
    opts).then((mongoose) => {
        return mongoose;
    });
}
cache.conn = await cache.promise;
return cache.conn;
}

export default connectToDB;

```

So, we have created a utility function `connectToDB` to handle the connection to MongoDB using Mongoose. The function (a singleton pattern) ensures that a single connection is maintained across the app, optimizing DB interactions.

4.4.2. Data Management and Storage

In the Next.js project, data from MongoDB can be fetched in API routes or directly within `getServerSideProps` or `getStaticProps` functions. This ensures that data is available during server-side rendering. It enables the pre-rendering of pages with dynamic content.

Example: Fetching Data in `getServerSideProps`

```

// pages/posts.js
import React from 'react';
import connectToDatabase from '../lib/mongodb';
import Post from '../models/Post';

const PostsPage = ({ allPosts }) => {
    return (
        <p>
            <h2>The posts</h2>
            <ul>
                { allPosts.map((onePost) => (
                    <li key={post._id}>{onePost.title}</li>
                ))}
            </ul>
        </p>
    );
};

```

```

export async function getServerSideProps() {
  const database = await connectToDatabase();
  const posts = await Post.find({}).lean();

  return {
    props: {
      posts: JSON.parse(JSON.stringify(posts)),
    },
  };
}

export default PostsPage;

```

So in this example, the `PostsPage` component fetches data from MongoDB using `getServerSideProps`. After that, the fetched posts are passed as props to the component. Then they are rendered on the server before being sent to the client. This way the page is pre-rendered with the data from the DB.

By integrating MongoDB with Next.js, developers can manage and store the app's data, enabling the development of dynamic and data-driven web applications.

5 Deployment and Maintenance

Deploying a web app is a crucial step in the development process. And in this chapter, we will explore what we need to do to prepare a Next.js app for deployment. We will look at different deployment strategies and best practices. (Vercel, 2024). The chapter will look at environment configuration, build optimizations, CI/CD, and monitoring practices to ensure the application remains performant and reliable.

5.1 Preparing the Application for Deployment

Before deploying a Next.js app, we need to do a few things to ensure its production-ready (Flanagan, 2020). Mainly we need to configure the environment variables and optimize the build for better performance.

5.1.1 Environment Configuration

We can define environment vars in Next using `.env` files (Smith & Johnson, 2019). They are very important for handling different settings in different

environments (dev, stage, prod). They can be accessed in the application using the `process.env`.

Example: Configuring Environment Variables

Create a `.env` file in the root of your project:

```
# .env
NODE_ENV = production
MONGODB_URI = mongodb://localhost:27017/myDB
API_KEY = your_api_key
```

Next.js automatically loads these variables and makes them available via `process.env`:

```
// pages/api/data.js
export function handler(request, result) {
  const apiKey = process.env.API_KEY;

  result.status(200).json({ apiKey });
}
```

By configuring environment vars, sensitive information such as API keys and database URLs can be securely managed and accessed in the app.

5.1.2. Build Optimization

Next.js gives us different optimization techniques to ensure the app is robust and efficient (Wilson & Harbison, 2018). Running the next build command generates an optimized production build of the app. That includes features like code splitting, minification, and static file serving.

Example: Building the Application

```
npm run build
```

The build process consists of several steps:

- **Code Splitting:** it splits the code into smaller bundles, which are then loaded on demand.
- **Minification:** it reduces the size of the JS files by removing unnecessary characters and whitespace. It makes the file unreadable for humans, but fine for the computer.
- **Image Optimization:** Optimizes images using the built-in Image component, which supports lazy loading and responsive images.

So if we improve the build, the apps will load faster and perform better. All of this gives a better overall user experience (Clark, 2021).

5.2. Deployment Strategies

We can use various strategies to deploy Next.js app. It depends on the hosting provider and the specific requirements of the project. So now we will explore deployment options such as Vercel and custom server deployments.

5.2.1. Vercel

Vercel is the company behind Next.js. It offers a mainstream deployment process for Next.js apps. By connecting a GitHub repository to Vercel, developers can automatically deploy their applications on every push (Smith, 2024).

Example: Deploying to Vercel

1. Sign up at <https://vercel.com/>.
2. Connect your GitHub repo to Vercel.
3. Vercel will then detect the Next.js project and configure the deployment settings automatically.
4. When you push to the main branch, Vercel will build and deploy the app.

Vercel provides additional features such as custom domains, environment variables management, and built-in analytics, making it a powerful platform for deploying Next.js applications (Flanagan, 2020).

5.2.2. Custom Server Deployment

There are cases where we need even more control over the deployment environment. Custom servers can be used to deploy Next.js apps. To do this, we need to set up a Node.js server, serving the Next.js app and configure a reverse proxy (can be Nginx) to handle incoming requests.

Example: Deploying on a Custom Server

1. Set up a Node.js server:

```
// server.js
const dev = process.env.NODE_ENV !== 'production';

const express = require('express');
const next = require('next');

const application = next({ dev });
const handle = application.getRequestHandler();

application.prepare().then(() => {
  const server = express();

  server.all('*', (request, result) => {
    return handle(request, result);
  });
  server.listen(5000, (error) => {
    if (err) throw error;
    console.log('> Ready on http://localhost:5000');
  });
});
```

2. Install dependencies and start the server:

```
npm install express
node server.js
```

3. Configure Nginx as a reverse proxy:

```
server {
  listen 80;
  server_name thedomain.co.uk;

  location / {
    proxy_pass http://localhost:5000;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection 'upgrade';
    proxy_set_header Host $host;
    proxy_cache_bypass $http_upgrade;
  }
}
```

There is greater flexibility in choosing the option to deploy to a custom server. We can configure the server and its resource management. Thus giving us a custom deployment setup

5.3. Continuous Integration & Continuous Deployment (CI/CD)

By implementing CI/CD pipelines we could automate testing, building, and the deployment processes. This will ensure that the changes are reliably integrated and deployed. Tools like GitHub Actions, CircleCI, and Jenkins can be setup to run tests, build the app, and deploy it to the server.

Example: Setting Up CI/CD with GitHub Actions

1. Create a GitHub Actions workflow file:

```
# .github/workflows/deploy.yml
name: Deploy

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2

      - name: Install dependencies
        run: npm install

      - name: Build the project
        run: npm run build

      - name: Deploy to Vercel
        run: vercel --prod --token ${ secrets.VERCEL_TOKEN }
```

2. Add the Vercel token to GitHub Secrets:

- Go to the GitHub repo settings.

- Then go to "Secrets and variables" > "Actions".
- Add a new secret with the name VERCEL_TOKEN and your Vercel token as the value.

So, if we use this setup, every push that goes to the main branch will automatically trigger the workflow. It then will install the dependencies, build the project and deploy it to Vercel. It's a very practical way of making pushes to the main branch.

5.4. Application Maintenance and Monitoring

Once the deployment is done, we need to maintain and monitor the app. Reliability and performance is crucial. In next section, we shall look at the best practices for error tracking, performance monitoring and making sure that the app stays up-to-date.

5.4.1. Error Tracking

We can use tools like LogRocker or Sentry. They can be integrated with Next.js to track & report errors in real time (MongoDB, 2024). It will help us find and fix any upcoming issues quickly.

Example: Integrating Sentry for Error Tracking

1. Install the Sentry SDK:

```
npm install @sentry/nextjs
```

2. Configure Sentry in your Next.js application:

```
// sentry.client.config.js
import * as Sentry from '@sentry/nextjs';

Sentry.init({
  dsn: process.env.SENTRY_DSN,
  tracesSampleRate: 1.0,
});
```

3. Capture errors in API routes:

```
// pages/api/data.js
import * as Sentry from '@sentry/nextjs';

export default function handler(request, res) {
  try {
    // Your code here
  } catch (error) {
    Sentry.captureException(error);
    res.status(500).json({ error: 'A error has occurred' });
  }
}
```

By using Sentry, developers can monitor errors & performance issues, making sure that they are swiftly addressed (Flanagan, 2020).

5.4.2. Performance Monitoring

By using a service like “New Relic & Datadog” we can monitor the performance of a Next.js app (Wilson & Harbison, 2018). The tools give us insights to the server response times, resources used, and other important data. It allows to optimize the app for performance.

Example: Integrating New Relic for Performance Monitoring

1. Install the New Relic Node.js agent:

```
npm install newrelic
```

2. Configure New Relic:

```
// newrelic.js
exports.config = {
  app_name: ['Your Next.js App'],
  license_key: 'your_new_relic_license_key',
  logging: {
    level: 'info',
  },
};
```


3. Start the application with New Relic:

```
NEW_RELIC_NO_CONFIG_FILE = true NEW_RELIC_APP_NAME = "Your
Next.js App" NEW_RELIC_LICENSE_KEY = "your_new_relic_license_key"
node -r newrelic server.js
```

There is no real downside to using performance monitoring tools. We can gain valuable information on the the app performs in prod and find any bottlenecks and areas for improvement (Smith & Johnson, 2019).

5.4.3. Keeping Dependencies Up-to-Date

Dependencies will need to be updated from time to time. It's a tricky endeavor because it can cause breaking changes. Sometimes one dependency can depend on a particular version of another so updating can be hard, but it's important. We can use a tool called Dependabot to automate the process.

Example: Setting Up Dependabot

1. Create a Dependabot configuration file:

```
# .github/dependabot.yml
version: 2
updates:
  - package-ecosystem: 'npm'
    directory: '/'
    schedule:
      interval: 'daily'
```

Using the above setup, Dependabot will check for outdated dependencies automatically and create pull request in order to update them. Our app will remain secure and up-to-date with the recent security patches (Smith & Johnson, 2019).

5.5. Conclusion

The whole process of deploying a Next.js app consists of a few main steps. Configuring the environment, build optimization, implementing CI/CD pipelines, and monitoring performance. We need to use the appropriate tools and existing best practices. This way we can ensure the application is reliable, performant and easy to maintain or update.

The chapter is a detailed guide for preparing, deploying, and maintaining a Next.js application. We integrated important technologies and used best practices to ensure the application remained scalable, easy to deploy and provided a solid foundation for future development (MongoDB, 2024).

6 Summary

In this thesis, we explored different tools, processes, and techniques to build a modern web app using Next.js. To this day, it is the single most popular framework for React. It allows for server-side rendering, which greatly improves load times and the SEA of the application. It was a journey that started from understanding the basics to using CI/CD to deploy and monitor the whole process. In the next chapter, we will highlight the main points and advantages of Next.js.

6.1. Overview of Next.js

Next.js gives us with a powerful toolkit for building modern web apps. Using the best aspects of React with server-side rendering capabilities, it offers a variety of features including:

- **Server-Side Rendering (SSR):** Improves load times & SEO because the pages are rendered on the server.
- **Static Site Generation (SSG):** Generates static HTML at build time, enabling faster page loads.
- **API Routes:** Allows the creation of backend endpoints directly within the Next.js application.

- **Dynamic Routing:** Facilitates building dynamic and nested routes with file-based routing.
- **Built-in CSS and Sass support:** Enables styling with CSS-in-JS, global CSS, and preprocessed stylesheets.

All of these features make Next.js a great choice for developing efficient and scalable web apps.

6.2. Key Components and Technologies

Throughout this thesis, we have integrated different technologies to build a web application:

- **React:** For building an interactive UI.
- **MongoDB:** A NoSQL database.
- **Node.js:** For creating the backend.
- **Express.js:** As a lightweight web app framework for handling server logic.

Each technology was selected to complement Next.js and enhance the application's functionality and performance.

6.3. Development Process

The development process involved several critical steps:

6.3.1. Setting Up the Project

- **Project Initialization:** We created a new Next.js project and configured the development environment.
- **Environment Configuration:** Setup environment variables for different deployment stages (dev, stage, prod).

6.3.2. Building the Application

- **Data Modeling:** Defined MongoDB schemas and models.
- **API Routes:** Created API routes
- **User Authentication:** Implemented user authentication using JWT and bcrypt.
- **Component Development:** Built reusable React components for the user interface.

6.3.3. Deployment

- **Build Optimization:** Ran next build to optimize the application for prod.
- **Deployment Strategies:** Deployed the application to Vercel.
- **CI/CD:** Set up CI (continuous integration) & deployment pipelines to automate the testing and deployment.

6.4. Challenges and Solutions

During the development process, several challenges were encountered:

- **Performance Optimization:** Ensured fast load times by using Next.js features like SSR and SSG.
- **Security:** Implemented secure authentication mechanisms and protected sensitive data using environment variables.
- **Scalability:** Designed the application architecture to handle increasing user load and data volume.

6.5. Advantages of Using Next.js

Using Next.js has given us numerous benefits:

- Server-side rendering gives better SEO visibility.
- Static site generation & optimized builds result in faster page loads.
- Development features and great documentation made the development process easier and more efficient.
- Next.js's architecture supports scalability for future growth and feature expansion.

6.6. Future Work and Recommendations

There are a few areas that could have the potential for future exploration:

- Advanced State Management using Redux
- GraphQL Integration for more efficient data fetching
- Implementing testing using Jest & Cypress.
- Adding monitoring and analytics to track application performance in prod.

6.7. Conclusion

Next.js is a powerful framework that provides us with even more advantages than using only regular React. It is a powerful and efficient way to create scalable and more performant apps. We can use SSR, SSG and a whole ecosystem of tools and libraries. The thesis has given us a detailed guide throughout the entire process. Starting with project setup and ending with deployment and maintenance.

During the process of developing “NextBlog”, we have demonstrated the advantages of Next.js. The benefits of using SSR, SSG, CI/CD, etc. Web technologies are constantly evolving and frameworks like Next.js play a crucial role in this process. It gives developers the tools they need for building even more powerful web applications.

References

1. Flanagan D. JavaScript: The Definitive Guide. O'Reilly Media; 2020.
2. Wilson E, Harbison SP. Node.js 8 the Right Way: Practical, Server-Side JavaScript That Scales. Pragmatic Bookshelf; 2018.
3. MongoDB. MongoDB Documentation [Internet]. <https://docs.mongodb.com/>
4. Express.js. Express.js Documentation [Internet]. <https://expressjs.com/>
5. React.js. React Documentation [Internet]. <https://reactjs.org/docs/getting-started.html>
6. Smith J, Johnson R. Scaling MongoDB for Large Datasets. J Database Manag. 2019;30(2):45-62. doi:10.4018/JDM.2019040103
7. Vercel. Vercel Documentation [Internet]. <https://vercel.com/docs>
8. Coursera. Full Stack Web Development with React Specialization [Internet]. Available from: <https://www.coursera.org/specializations/full-stack-react>
9. Next.js Examples. Official Next.js GitHub Repository [Internet]. <https://github.com/vercel/next.js/tree/canary/examples>
10. Stack Overflow. Online Developer Community [Internet]. <https://stackoverflow.com/>
11. Clark C. Mastering Next.js: The Full Guide to Server-Side Rendering and Static Site Generation in Next.js. Packt Publishing; 2021.
12. Hall B. MongoDB Applied Design Patterns: Practical Use Cases with the Leading NoSQL Database. Packt Publishing; 2020.