Dinh Bui

NEXT.JS FOR FRONT-END AND COMPATIBLE BACKEND SOLUTIONS

Bachelor's thesis

Information Technology

Bachelor of Engineering

2023

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland
University of Applied Sciences

Degree title	Bachelor of Engineering
Author(s)	Dinh Bui
Thesis title	Next.js for front-end and Compatible Backend Solutions
Commissioned by	XAMK
Year	2023
Pages	46 pages
Supervisor(s)	Timo Mynttinen

ABSTRACT

As a result of the rapid advancement of technology today, several web frameworks are being developed to facilitate better software development. One of the newest and rising in popularity frontend frameworks nowadays is Next.js. Many developers are looking for the easiest and most flexible backend solutions for their personal or professional projects since they are accustomed to handling the Next.js framework.

As a result of the increasing popularity of such a framework, the aim of this thesis was to identify Next.js-compatible backend alternatives. This thesis compared the benefits and drawbacks of two backend solutions based on a comprehensive scenario.

In this thesis, Next.js was used to create a web application for a scenario, which was designed to search for plants. The two solutions evaluated were Express.js with MongoDB and Firebase. They were installed with a database that stored data from an external API and they were configured to provide API endpoints for displaying contents on Next.js frontend.

The output of this thesis supported the evaluation of the advantages and disadvantages of backend solutions. Express.js with MongoDB provided greater flexibility and scalability. On the other hand, Firebase concentrated more on features like real-time databases, authentication, cloud storage, hosting, and analytics. It offered a wide range of services for quick development and monetization.

Keywords: Next.js, Express.js, MongoDB, Firebase

South-Eastern Finland
University of Applied Sciences

CONTENTS

1 INTRODUCTION

Nowadays, technologies are thriving so fast, which means many web frameworks are developed to support better software development. Frontend developers have many frameworks to choose from, such as Nuxt.JS, Angular, Vue, React, Next.js, etc. Any frontend developer would want to choose a framework that can assist them in creating a web application quickly and with minimal errors. One of the latest and most widely used frontend frameworks nowadays is Next.js. Many people are familiar with using Next.js and want to find the easiest and most compatible backend solutions for their personal or work projects. However, they do not know what the backend entails or if it can be a suitable match, and how to integrate it into their frontend.

Therefore, my goal in this thesis is to conduct research to identify suitable backend solutions for Next.js. Furthermore, I will compare the strengths and weaknesses of these backend solutions based on the scenario I create. The comparison will be based on various factors. Afterwards, developers can choose the appropriate solution for their project's specific purpose to reduce the likelihood of errors.

The thesis begins by presenting the theoretical background, which will cover the basic information and general definition of a web application. There is a more detailed theoretical explanation of Next.js and React.js and the comparison between them. The theoretical part includes some relevant terms for the implementation part. The structure will be described as follows:

Chapter 1 is the introduction to the general information as well as the purpose of the thesis.
Chapter 2 describes the definition of relevant terms and concepts of a web application.
Chapter 3 describes a scenario of choice with Next.js and provides a walk-through guide on how to integrate different backend solutions.

Chapter 4 is the summary of the result of the thesis and the recommendation for developers.

2 THEORY

This section presents the definition of the technologies used for the research. It also covers the concepts of web applications and the method that are required for the thesis.

2.1 Web application

The web application is used for various tasks, and it usually includes webmail, e-commerce shops, and transactions. Create dynamic apps that reflect a new form of cooperation and collaboration among many users thanks to new Web technologies, languages, and approaches. Component orientation and standard components are software engineering methodologies that web application development has quickly adopted (Jazayeri 2007). Web applications distinguish from websites in that web applications can modify data on the server (Facebook, Google Mail, etc.) and websites are used to watch and read the information on a landing page such as a newspaper (BBC News, YouTube, etc.). Web applications is an application program that is stored on a remote server. Users can access web applications through most web browsers (Google Chrome, Microsoft Edge, Safari, etc.) but there are some web apps that could be accessed by a specific browser.

Web applications are divided into 3 main parts which are a web server, an application server, and a database. This is how a web application process a request (Figure 1):

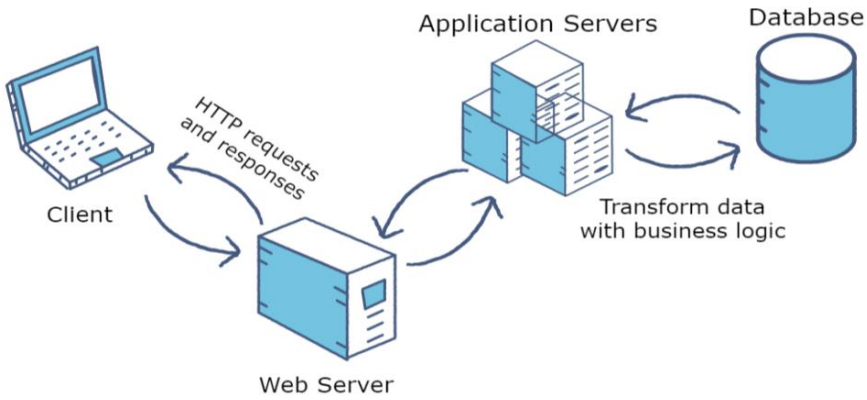

Figure 1 Operation of a web application

Users can use their web browser or the program's user interface to request access to a web application, and the web server receives the request through the Internet. The request will be sent by the web server to the application server. The application server is responsible for processing the request, which may include running database queries. After processing the data requests, the application server sent the result to the web server, which responds to the client with the requested information. This data will be displayed on the screen of the user through their web browser or application interface. This operation enables users to interact with web applications and retrieve information in an easy and efficient way.

2.1.1 Web application frontend

Frontend is part of a web application or website that users interact with directly. It is also known as client-side development. Frontend development involves creating the user interface of the website or web application, such as buttons, links, animations, and more. Frontend developers will use web technologies to build the website or web application such as HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), and JavaScript. The goal of frontend development is to create a visually attractive and interactive experience for the user. Some primary components of the frontend include:

- HTML – This is the basic building block for frontend development – used to create the structure and content of web pages.

- CSS – This is used to style the content of the HTML and create visual elements, such as fonts, color, and layout.
- JavaScript – This is a scripting programming language. It uses to add interactivity and dynamic functionality to web pages, such as animations, user input validation, and data manipulation.
- Frontend framework and libraries – These are pre-built components and tools that simplify and speed up frontend development, such as Angular, Vue.js, React, Next.js, Bootstrap, Materialize, and Tailwind CSS.

2.1.2 Web application backend

Backend development is knowns as server-side development. It is a crucial part of creating web applications. It involves designing and implementing the architecture of the application, creating APIs (Application Programming Interfaces) that allow the frontend of the application to interact with the backend, and managing data and content storage. Some of the key concepts and technologies involved in backend development:

- Server-side programming language: There are many programming languages that can be used, such as Python, Ruby, Java, or PHP to write the code that runs on the server. These languages are designed to handle complicated operations, data processing, and database management.
- Database: Data is an indispensable component of most web applications, and backend developers use databases to store and manage this data. Some popular databases used in backend development include MySQL, PostgreSQL, and MongoDB
- APIs: APIs are interfaces that use for communication between applications. Backend developers create APIs that allow backend and frontend of the application can interact with each other and perform various operations such as fetching data, submitting forms, and performing transactions.
- Frameworks: There are several frameworks for backend such as Django, Ruby on Rails, Spring, and Laravel that provide developers with pre-built tools and libraries to help speed up the development process. These frameworks

handle many of the routine tasks such as user authentication and data validation, allowing developers to focus on building more complex functionality.

- Cloud platform: Cloud platforms such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) provide backend developers with access to scalable infrastructure and services such as virtual servers, databases, and APIs. These platforms allow developers to deploy their applications quickly and easily to a global audience.

In contrast to frontend development, which focuses on the user-facing aspect of websites or applications, backend development is concerned with the underlying functionality that supports building frontend. Both frontend and backend development are critical components of building a successful web application or website.

2.2 Next.js Framework

Next.js is an open-source React frontend framework. Based on its documentation, Next.js is a new framework designed to solve React's limitation of building full server-side or server-side rendering. Next.js is built based on the React library, which means it takes advantage of React and adds several features. All of the packages and configuration files are neatly and logically bundled together using Next.js. It stands out from other web application frameworks since it offers developers a location to build both the frontend and backend code for an application. Consequently, it eases the developer's workload and aids in the quickest shipment of the finished product (Sasikumar et.al. 2022). It is designed to enable developers to build server-rendered React applications easily. Next.js offers many features, including Server-side rendering (SSR), Automatic code splitting, Static site generation (SSG), Client-side routing (CSR), CSS and Sass support, plugins and integrations.

- SSR: Next.js allows developers to render React components on the server, which can improve the initial loading time of the application. This means that

the HTML for the page is generated on the server and sent to the client, instead of having the client render the entire page. This approach can be especially useful for SEO purposes.

- Automatic code splitting: When the user navigates through the applicationNext.js will automatically split the JavaScript code into small blocks so that only the code needed for a particular page is loaded. This helps to reduce the initial load time of the application and improve performance.
- SSG: Next.js supports Static site generation, which generates HTML files for pages at build time. It helps to reduce the server load, improve performance, and more secure.
- CSR: When the user clicks on the link, the page loaded dynamically without a full-page refresh. This can improve the user experience and make the application feel more like a traditional web application.
- CSS and Sass support: Next.js has built-in support for CSS and Sass, making it easy to style the application. It also supports CSS modules, which helps to prevent CSS class name duplication and makes it easier to maintain your styles.
- TypeScript support: Next.js has built-in support for TypeScript, making it easier to build type-safe and clear applications.

Next.js is a powerful web development framework that offers so many features and benefits. It makes it easy to build fast, secure, scalable, and optimized web applications that can handle high traffic loads.

2.3 React

React is an open-source JavaScript library used for building frontend developments such as user interfaces (UIs) and single-page applications. It was developed by Facebook and released in 2013. It is currently maintained by Facebook and a community of individual developers and companies. React also provides code reusability by introducing two important definitions, JSX, and virtual DOM, to speed up the development process and reduce the risks that may occur while coding. According to its documentation, React can be used in many

contexts, from small websites to large-scale web applications. It can be used with JavaScript libraries and frameworks, such as redux and Next.js, to create complex dynamic applications. A previous study found that maintaining web application states and making API queries were marginally easier with React's capabilities (Sianandar & Manuaba 2022). React also has a large and active community of developers, which means that there are many resources and tools available to help developers learn and use the library effectively.

Next.js is a great framework built on top of React.js but there still have some key differences between Next.js and React.js:

- Routing: React.js does not have a built-in routing system, which means that developers need to use a third-party library such as React Router to handle client-side routing. On the other hand, Next.js provides a built-in routing system, that allows developers to define their routes using a file system-based approach. This means that developers can create new pages by simply creating a new file in the pages directory, and Next.js will automatically generate the necessary routes.
- SSR: With Next.js, Server-side rendering is readily available, allowing the server to create the initial HTML for each page and sent it to the client, rather than relying on client-side JavaScript to generate the content. This can have benefits for website performance, search engine optimization (SEO), and user experience, especially for users with slow or unreliable internet connections. On the other hand, React.js only supports CSR by default, which generates the page on the client side after the initial HTML is downloaded.
- Tooling: Next.js already has many tools that were built into the framework to help boost the development process, including static site generation, hot reloading, and automatic code splitting. These features save developers an amount of time and effort and make it easier to build complex web applications. On the other hand, React.js does not have built-in tools like Next.js so developers need to set up and configure their own tooling and environment, which can be time-consuming and difficult for beginners.

- Learning Curve: Next.js is built on top of React.js, developers who already have experience with React.js will find it easier to learn and work with Next.js. However, developers who are new to React.js may need to spend more time learning the basics before they can effectively use Next.js.
- Deployment: Next.js includes built-in support for serverless deployment on platforms such as Vercel, which can significantly simplify built-in support for the serverless deployment process. React.js applications require more manual configuration and setup in order to deploy them to a production environment. Hence, deploying a Next.js application is easier than deploying a React.js application, especially when taking the advantage of server-side rendering.

2.4 Backend solution: Express.js with Mongo DB

The backend solution is an essential part of a web application or software system that is responsible for processing data and managing the server-side logic. It includes a set of tools and languages that are used in programming. In the practical part of this thesis, Express.js with MongoDB and Firebase will be the solutions implemented in my scenario.

According to its website, Express.js is a flexible Node.js web application framework. It is used for designing and building web applications and APIs more quickly and easily. Some of the features of Express.js are the following:

- Routing: Express.js has a powerful routing API that enables programmers to create unique routes for dealing with HTTP requests. This makes it simple for developers to design web applications and RESTful APIs.
- Middleware: The Express.js framework includes middleware functions as a core component. They can be utilized to run programs either before or after a request is processed, to alter request or response objects, or to deal with problems.

- Express.js supports a number of different templating engines, including EJS, Pug, and Handlebars. As a result, rendering and serving dynamic HTML pages to clients is made simple for developers.
- Error Handling: Express.js comes with an error handling system that can be used to identify and address errors that occur during the request-response cycle.
- Static files serving: Express.js makes serving static files like pictures, CSS, and JavaScript simple. This could help in improving the responsiveness and performance of web applications.
- Middleware ecosystem: Many third-party middleware modules that expand the functionality of the Express.js framework have been created as a result of large and active developer and contributor communities.

MongoDB is a well-known example of an open-source document-oriented NoSQL database application. Based on its website, MongoDB has excellent performance, flexibility, and scalability, and it saves data in a JSON-like style. MongoDB is designed to handle massive volumes of data and gives programmers the ability to construct applications that can manage complex queries and data structures. It also enables indexing and has a robust query language. Each language has an official document on its website, which makes MongoDB becomes one of the most well-known databases in the world. The features of MongoDB include:

- High availability: MongoDB can be configured to run on many servers, which provides high availability and fault tolerance.
- Security: Security features built into MongoDB include role-based access control, authentication, and SSL/TLS encryption.
- Performance: MongoDB can handle large volumes of data and high write loads with ease.
- Indexing: MongoDB provides secondary indexes, which allows for faster queries.

- Aggregation: MongoDB includes a robust aggregation framework that can perform complex queries and data analysis.

Firebase is a mobile and web application development platform that offers a variety of services to assist developers to build, grow, and monetize their applications. According to its documentation, Firebase is a NoSQL database application that saves data in JSON-like documents. Some of the key features of Firebase include:

- Real-time Database: Firebase includes a real-time database that allows developers to store and sync data between many clients in real-time. This capability is especially beneficial for creating collaborative apps like chat apps, multiplayer games, and other real-time apps.
- Authentication: Firebase includes authentication capability that works with a variety of authentication providers, including email and password, phone number, Google, Facebook, Twitter, and GitHub. This feature enables developers to simply include user authentication in their applications without the need to design and maintain sophisticated authentication code.
- Cloud storage: Firebase offers a cloud-based storage solution that allows developers to immediately store and serve user-generated material, such as photographs and videos, from the Firebase console.
- Hosting: Firebase offers a scalable and robust hosting solution that allows developers to easily deploy and serve their applications with a single command. Custom domains, SSL certificates, and automatic integration with their content delivery networks (CDNs) are also supported.
- Analytics: Firebase offers an extensive analytics solution that gives developers insights into user activity, engagement, and retention, among other things.

3 PRACTICAL IMPLEMENTATION

Two of the most popular backend solutions for Next.js that I have mentioned in the theory part are Express.JS with MongoDB and Firebase. Expres.JS with

MongoDB is the most known for building restful APIs and being customizable, whilst Firebase is a service, which provides a more streamlined and managed backend solution. In this part, I compare both solutions based on these criteria: functionality, ease of use, and documentation (community)

Functionality: A solution is nothing without its functionalities. To dig deeper into this, I would build a scenario, in which I would set up and integrate each solution to my Next.js application as well as test out their basic functionalities, such as making an API call to an external service, providing API endpoints for the frontend, writing to the database and reading from the database.

Ease of use: One of the decisive factors in choosing any software solution is how easy it is for developers to implement and to scale. Any difficulty I met while implementing would be displayed and explained and after that, I would give my thoughts on how the whole process feel like for each solution and research on how scalable they are.

Documentation (community): Documentation as well as the community that supports each backend solution is always important. They keep the solution up-to-date, safe, and easy to use, which are of the most important traits of any software application/service. Any instruction or help I could find while solving the errors during the implementation phase would dictate how good, detailed, or supportive the documentation and community are behind each backend solution.

3.1. Scenario

The idea I had in mind for this scenario was to create a simple “Wikipedia” search for house plants, where users can come and enter the name of the plant and my application would display information about the plant in search. The public plant API I chose is from Perenual©. The custom-built REST-ful API offers information about over ten thousand plant species. I was on the free subscription of the API service and I only needed one API endpoint for my application, which is https://perenual.com/api/species-list?page=1&key=[YOUR-API-KEY]. It would provide data about all plants in their database in JSON format.

The application would be using Next.js with Typescript for the frontend. For comparing backend solutions, I would create two repositories with the same frontend code, one for Express.JS with MongoDB, and one for Firebase. In doing that, I would be able to test the following:

- Backend set up: how to integrate each solution to a Next.js application.
- Write to database: Calling the API to get all plants and their information, and then store them in the database of each solution.
- Read from database: When user enters a name of a plant, my application will send a query to the database, fetch the information in JSON format and displays it on screen.

The steps I described below to create the Next.js application would be the same for both repositories, their name would be plant-next-js and plant-next-js-firebase. First, I initialized Next.js web application with the command npx create-next-app and these are the settings I chose:

```
User:~ user$ npx create-next-app

✓ What is your project named? … plant-next-js
✓ Would you like to use TypeScript with this project? … No / Yes
✓ Would you like to use ESLint with this project? … No / Yes
✓ Would you like to use Tailwind CSS with this project? … No / Yes
✓ Would you like to use `src/` directory with this project? … No / Yes
? Would you like to use experimental `app/` directory with this project? › No / ✓ Would you like to use experimental `app/` directory with this project? … No / Yes
✓ What import alias would you like configured? … @/*
Creating a new Next.js app in /Users/dinhbuimac/plant-next-js.
```

Once the app was created, I started the app with the command:

```
User:plant-next-js dinhbuimac$ npm run dev

> plant-next-js@0.1.0 dev
> next dev
```

```
ready - started server on 0.0.0.0:3000, url: http://localhost:3000
event - compiled client and server successfully in 1259 ms (167 modules)
wait  - compiling...
event - compiled successfully in 86 ms (134 modules)
```

Then, the application can be seen at http://localhost:3000 (Figure 2).

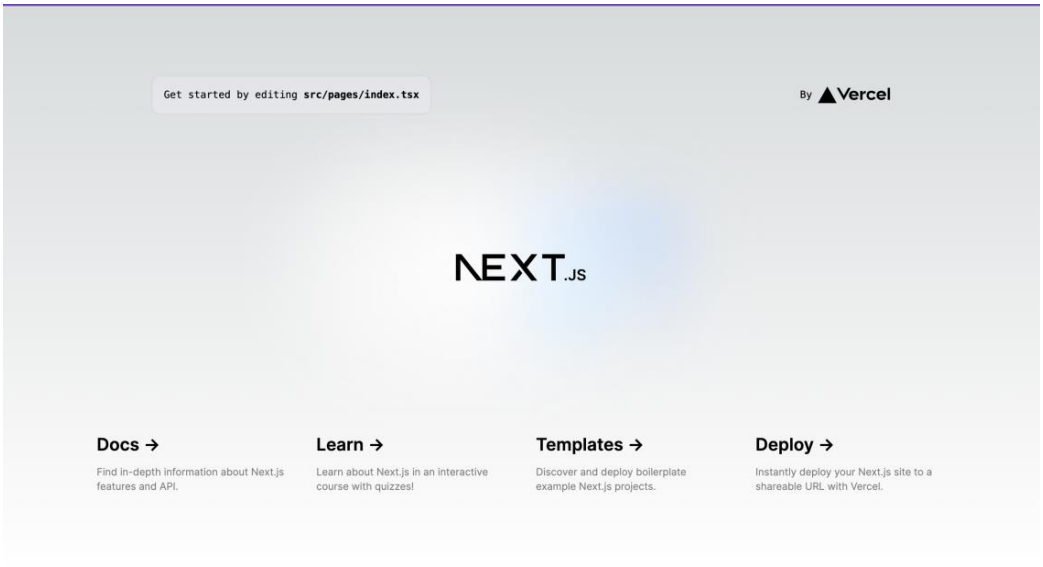

Figure 2 Next.js application initiation

Secondly, I imagined my front page would have a simple search bar for users to type in the name. I modified the function Home in index.tsx file to have a button and a search bar in the middle of the page:

```
import { Inter } from 'next/font/google'

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

export default function Home() {
  return (
    <main className={`flex min-h-screen flex-col items-center justify-between p-24 ${inter.className}`}>
      <h1 className="mb-4 text-4xl font-extrabold leading-none tracking-tight text-gray-900 md:text-5xl lg:text-6xl dark:text-white">Plants <mark className="px-2 text-white bg-green-800 rounded dark:bg-green-700">Wikipedia</mark></h1>
```

```
      <div className="flex items-center">
       <button className="bg-green-800 hover:bg-green-700 text-white font-bold py-2 px-4 rounded">
        Update all plants
       </button>
       <div className="relative ml-3">
        <div className="absolute top-3 left-3 items-center">
         <svg className="w-5 h-5 text-gray-500" fill="currentColor" viewBox="0 0 20 20"
xmlns="http://www.w3.org/2000/svg"><path fillRule="evenodd" d="M8 4a4 4 0 100 8 4 4 0 000-8zM2 8a6 6 0
1110.89 3.476l4.817 4.817a1 1 0 01-1.414 1.414l-4.816-4.816A6 6 0 012 8z"
clipRule="evenodd"></path></svg>
        </div>
        <input
          type="text"
          className="block p-2 pl-10 w-70 text-gray-900 bg-gray-50 rounded-lg border border-gray-300"
          placeholder="Search Here..."
        />
       </div>
      </div>
    </main>
  )
}
```

I also added a background image to the body of the web page in styles/global.css file so that the home page would look more refined (Figure 3).

```
body {
  color: rgb(var(--foreground-rgb));
  background-image: url("/tropical-leaves-background-zoom_52683-40995.avif");
  background-repeat: repeat;
  width: 100%;
  background-size: contain;
}
```

Figure 3 The frontend of the application

Figure 3 showed the interface of my application using Next.js. The application is used to find information about plants.

3.2. Backend integration

In this part, I will install the database and the collection before fetching data from the external API. Then, I will test then Express.js with MongoDB and Firebase with my Next.js application.

3.2.1 Express.js and Mongo DB

First, I would need to create a MongoDB on my computer. To install MongoDB easier, I used Homebrew, which is a package manager for MacOS. This was the command I used in the terminal:

brew tap mongodb/brew

brew install mongodb-community

When MongoDB was installed, it needed a data directory in my computer to store its data. To set that up, I created a file named db in the data directory by running the command:

```
sudo mkdir -p ~/data/db
```

Since I used Macbook, MacOS already reserved /data/db path so I had to use ~/data/db instead. Then, I had to give MongoDB permissions to access and modify the data directory. Otherwise, MongoDB would raise an error that it could not find the designated directory.

```
sudo chown -R `id -un` ~/data/db
```

Next, I started the MongoDB server with command `mongod --dbpath ~/data/db`. The `--dbpath` option was to point MongoDB to the correct data directory that it should be using. MongoDB's default port to listen was 27017. I would keep this `mongod` terminal running throughout the practical part as to keep MongoDB service running always. Then, in another terminal tab, I used the command `mongosh` to open the mongo shell and interact with mongoDB. Once connected to MongoDB, I created a database for my Next.js web application, which was named plants.

```
test> use plants
switched to db plants
```

The database could only be created for the first time when I inserted data into it. So, I created a user name Dinh and store it to a "user" collection in the database.

```
plants> db.user.insert({name:"Dinh"})
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or
bulkWrite.
{
  acknowledged: true,
```

```
  insertedIds: { '0': ObjectId("644ea5a80ca9fa609d8a85f8") }
}
```

Once that was done, I could check if the database was created correctly with the command show dbs, which will list all the MongoDB databases in my computer.

```
plants> show dbs
admin   40.00 KiB
config  108.00 KiB
local   72.00 KiB
plants   8.00 KiB
```

Then, I created a collection named ”plants” to store and provide data for my application with the command db.createCollection("plants"). Afterwards, I called show collections (list all collections inside “plants” database) to see if it was created correctly.

```
db.createCollection("plants")
{ ok: 1 }
plants> show collections
plants
user
```

After MongoDB database and collection were set up, I would then add Express.Js and MongoDB to my Next.js web application. My Next.js web application should be connected to MongoDB through the help of Express.JS, so that it could write data to the database as well as retrieve data.

First, I installed the necessary libraries, which are Express.js and MongoDB by using the command:

```
npm install express mongodb @types/express @types/mongodb
```

Next, I installed ‘axios’, which is used to send HTTP requests to retrieve data from an API external or to send data to a server for processing. I added the ‘dotenv’ module to my application, which is used to load environment variables from a ‘.env’ file into the Node.js ‘process.env’ object. It allows me to store sensitive information, such as API keys or database passwords, outside of the codebase and in a separate file that is not shared in version control.

```
npm install axios
```

```
npm install dotenv
```

Second, I created a file named ‘server.ts’ in the root directory of the project for the Express.js server. At the beginning of the file, I imported external libraries, which are the express, MongoDB, and Axios functions into the file.

```
const express = require('express');
const { MongoClient } = require('mongodb');
const axios = require('axios');
require('dotenv').config()
```

Then, I initialized a new Express.js, and MongoClient instances and stored them as constants. The URI constant defined the MongoDB connection string, which was pointing to a local MongoDB server running on the default port 27017 and connecting to a database named “plants”.

```
const app = express();
const uri = 'mongodb://localhost:27017/plants';
const client = new MongoClient(uri);
```

After that, I used an asynchronous function called “connect” that uses the MongoClient instance created previously to connect to the MongoDB database. The `await` keyword was used to wait for the `client.connect()` method to establish a connection to the MongoDB database before proceeding. Once the connection is established, the message ’Connected to MongoDB’ is logged to the console,

which means the connection was successful. If there is an error connection to the database, the error message is logged to the console instead. I used the connect() to call the connect function, which would initiate the connection to the MongoDB database.

```
async function connect() {
  try {
    await client.connect();
    console.log('Connected to MongoDB');
  } catch (err) {
    console.log('Error connecting to MongoDB:', err);
  }
}
connect();
```

Next, I set up an API endpoint by using 'app.post()'. It listened for POST requests to 'api/plants'. Then, I used client.db().collection('plants') that was used to connect to the MongoDB database 'plants' and collections 'plants'. To avoid the data would be duplicated in the collection, I used await plantsCollection.deleteMany({}) for clearing out the 'plants' collection before populating it with new data from an external API.

```
app.post(`/api/plants`, async (req: any, res:any) => {
  try {
    // connect to MongoDB database "plants" and collection "plants"
    const plantsCollection = client.db().collection('plants');
    // deplete every plants inside the collection
    await plantsCollection.deleteMany({});
```

This API has a limit of 30 trees per page, which is insufficient for my testing needs. Therefore, I implemented a loop to fetch data from page 1 to 99 to collect enough data for testing. After that, I used the command axios.get(`${process.env.API_BASE_URL}?page=${i}&key=${process.env.API_KEY}`) to make a GET request to the external API to get the plants data. I also had to create the .env file, which contains variables such as 'API_BASE_URL' and 'API_KEY'.

```
API_BASE_URL=https://perenual.com/api/species-list
```

```
API_KEY= <MY_API_KEY>
```

Those variables are used in the 'axios' function call to make a request to an external API. Then data from the external plant API would be parsed and stored it in the plants variable. Then, I inserted the plant documents into the plant collection in the MongoDB database by using 'await lantsCollection.insertMany(plants)'.

```
    // make get request to plant API and write all plants to database
    for(let i = 1; i <= 99; i++) {
        console.log(i)
        const response = await
axios.get(`${process.env.API_BASE_URL}?page=${i}&key=${process.env.API_KEY}`);
        const plants = JSON.parse(JSON.stringify(response.data.data));
        await plantsCollection.insertMany(plants);
    }
```

If an error occurs during the processing, the catch block will log the error to the console and send the error response back to the client.

```
    console.log("Plants collection populated with data from API")
    res.send("Plants collection populated with data from API");
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

After the data is populated to the collection, I created a function to define an HTTP GET endpoint for search plants by name. When the endpoint is requested with a plant name as a parameter (`/api/plants/:plantName`), the code retrieves the plant data from a MongoDB database, filters the plants whose common name contains the provided 'plantName' using a regular expression, and sends the filtered plant data as a JSON response to the client.

```
app.get(`/api/plants/:plantName`, async (req: any, res:any) => {
    const plantName = req.params.plantName;
    try {
```

```
    const plantsCollection = client.db().collection('plants');
    const regex = new RegExp(plantName, 'i');
    const plants = await plantsCollection.find({ common_name: regex }).toArray();
    res.json(plants);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

I have configured the port number of the server and started listening for incoming HTTP requests. I used the listen() method to call on the 'app' object, which is an instance of the Express.js application. The callback function passed to 'listen()' is executed when the server starts listening for incoming requests. I logged a message to the console when the server is running and ready to accept incoming requests.

```
const PORT = process.env.PORT || 5001;
const server = app.listen(PORT, () => {
 const serverAddress = server.address();
 console.log(`Server running at http://${serverAddress.address}:${serverAddress.port}`);
});
```

After configuring everything in server.ts file, I opened the package.json file to add two commands to the scripts: "dev" and "server". The "dev" script starts the development server for the Next.js application by running the command npm run dev. The "server" script starts the server-side TypeScript code using 'ts-node'. Since the server-side code was written in TypeScript, it needs to be compiled to JavaScript before it can be executed by Node.JS. I used the command npm run server to start the server-side TypeScript code using 'ts-node'.

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

```
  "server": "ts-node server.ts"
},
```

Then, I tested the development side and the server side and Both Next.js and server.js were running successfully (Figure 4 and Figure 5).

```
Dinhs-MacBook-Pro:plant-next-js dinhbuimac$ npm run dev

> plant-next-js@0.1.0 dev
> next dev

ready - started server on 0.0.0.0:3000, url: http://localhost:3000
info  - Loaded env from /Users/dinhbuimac/Documents/plant-next-js/.env
event - compiled client and server successfully in 2.6s (174 modules)
wait  - compiling...
event - compiled successfully in 113 ms (141 modules)
wait  - compiling /_error (client and server)...
event - compiled client and server successfully in 59 ms (175 modules)
warn  - Fast Refresh had to perform a full reload. Read more: https://nextjs.org/docs/messages/fast-refresh-reload
wait  - compiling / (client and server)...
event - compiled client and server successfully in 774 ms (246 modules)
```

Figure 4 The server testing in Next.js

```
Dinhs-MacBook-Pro:plant-next-js dinhbuimac$ npm run server

> plant-next-js@0.1.0 server
> ts-node server.ts

Server running at http://:::5001
Connected to MongoDB
```

Figure 5 The server testing in Next.js output

Next, I added the function for the button, which I created previously in the index.tsx file to update plants to the database. I created an asynchronous function called a POST endpoint in the application API (`/api/plants`) to update the database with plant data from the external API. When the function is called, it makes an HTTP POST request the the 'api/plants' endpoints using the Axios library. If the request is successful, the server responds with a message indicating that the database has been updated ('Plants collection populated with data from API')

```
async function updateAllPlants() {

  try {
```

```
    const response = await axios.post(`/api/plants`);
    console.log(response.data); // Updated plants database!
  } catch (error) {
    console.error(error);
  }
}
```

I created a components folder and inside that folder, I created the card.tsx. This file is used to render the information about plants and to perform the search function using the plantName.
First, I imported the necessary libraries for my component. Especially, the 'Image' library I added,

```
import React from 'react';
import axios from 'axios';
import Image from 'next/image';
import ReactDOM from 'react-dom';
```

I defined two TypeScript interfaces: 'PlantData' and 'CardProps. The 'PlantData' interface defined the structure of an object representing data. It has several properties, such as 'id', 'common_name', scientific_name', and 'default_image'. The 'CardProps' interfaces defined the type of data expected to be passed to a React component that renders a plant card.

```
interface PlantData {
    id: number;
    common_name: string;
    scientific_name: string[];
    other_name?: string[];
    default_image: {
      original_url: string;
      small_url: string;
      medium_url: string;
      thumbnail: string;
      license: number;
      license_name: string;
      license_url: string;
    };
```

```
  sunlight: string[];
  watering: string;
  cycle: string;
 }

interface CardProps {
 plantData: PlantData;
}
```

For the styling part, I mainly used Tailwind CSS. I searched the internet for a Tailwind CSS template for cards and buttons. I changed the plant information as needed.

```
const Card: React.FC<CardProps> = ({ plantData }) => {
 return (
  <div className="card-container">
    <div className="max-w-sm rounded overflow-hidden shadow-lg bg-white">
      <Image width={500} height={500} className="w-full" src={plantData.default_image.medium_url}
alt={plantData.common_name} />
       <div className="card-body px-6 py-4">
         <div className="font-bold text-xl mb-2">{plantData.common_name}</div>
         <p className="text-gray-700 text-base">Scientific name: {plantData.scientific_name}</p>
         <p className="text-gray-700 text-base">Cycle: {plantData.cycle}</p>

       </div>
       <div className="card-body px-6 py-4">
        <div white-space={"nowrap"}>
         <span className="inline-block bg-gray-200 rounded-full px-3 py-1 text-sm font-semibold text-
gray-700 mr-2 mb-2">
           Sunlight: #{plantData.sunlight}
         </span>
        </div>
        <div white-space={"nowrap"}>
         <span className="inline-block bg-gray-200 rounded-full px-3 py-1 text-sm font-semibold text-
gray-700 mr-2 mb-2">
           Watering: #{plantData.watering}
         </span>
        </div>
```

```
            {/* {plantData.tags.map((tag, index) => (
            <span key={index} className="inline-block bg-gray-200 rounded-full px-3 py-1 text-sm font-
semibold text-gray-700 mr-2 mb-2">
              #{tag}
            </span>
            ))} */}
          </div>
        </div>
        <br></br>
    </div>
  );
};
```

Then, I created an ‘async’ function called handeSearch that is called when the user presses the Enter key in the search input field. If the Enter key was pressed (keycode === 13), I sent the GET request to the server at the endpoint `/api/plants/${plantName}` where: plantName is the name of the plant the user is searching for. If the request is successful, the response data is an array of objects that contains information about the plants that match the search query. The code would map over this array to create a list of ‘Card’ components, passing in the information for each plant as a prop.

```
export async function handleSearch(event: any) {
  if (event.keyCode === 13) {
    // The Enter key was pressed, so get the plant name from the input field
    const plantName = event.target.value;

    // perform the search using the plantName
    try {
      const response = await axios.get(`/api/plants/${plantName}`);
      const plantDataList = response.data;
      console.log(plantDataList)
      const cards = plantDataList.map((plantData: any) => <Card key={plantData.id} plantData={plantData}
/>);
```

I used the ‘ReactROM.render’ to render the list of ‘card’ components to the ‘div’ element with the id ‘plantCard’ on the webpage.

```
    ReactDOM.render(<div>{cards}</div>, document.getElementById('plantCards'));
  } catch (error) {
    console.error(error);
  }
 }
}
```

When testing the event search feature, I encountered a 404 error in the console, which indicated that I could not connect to the server. After investigating the issue, I discovered that it was caused by the proxy used in Next.js, which differs from React.js. Fortunately, the supportive Next.js community helped me resolve the issue. To fix the problem, I added the 'rewrites()' function to next-config.js. This function defines a single rewrite rule that redirects incoming requests matching the pattern 'api/:path*' to 'http://localhost:5001/api/:path*', effectively configuring a proxy that forwards requests to a backend server running on 'localhost:5001'

```
 async rewrites() {
  return [
   {
    source: '/api/:path*',
    destination: 'http://localhost:5001/api/:path*'
  ]
 }
}
```

Figure 6 presents the final Next.js application with Express.js and MongoDB. The search engine was able to find multiple types of roses when the keyword was entered into the search bar.

Figure 6 Final Next.js application with Express.js and MongoDB

Figure 6 represents the result of the rose tree information I searched for by using Express.js with MongoDB. Detailed plant information is displayed, such as the scientific name, growth cycle, sunlight requirements, and more. The database contains various types of roses, allowing users to search for the specific tree type they need, enabling convenient plant care.

3.2.2. Firebase

I used the second repository to run with Firebase in the Firebase part. Before using Firebase, I had to create an account through this link ‘https://console.firebase.google.com/ ‘. Then I created a new project by clicking on the “Add project” button and choosing the default setting option and the name of the project is “plants” (Figure 7).

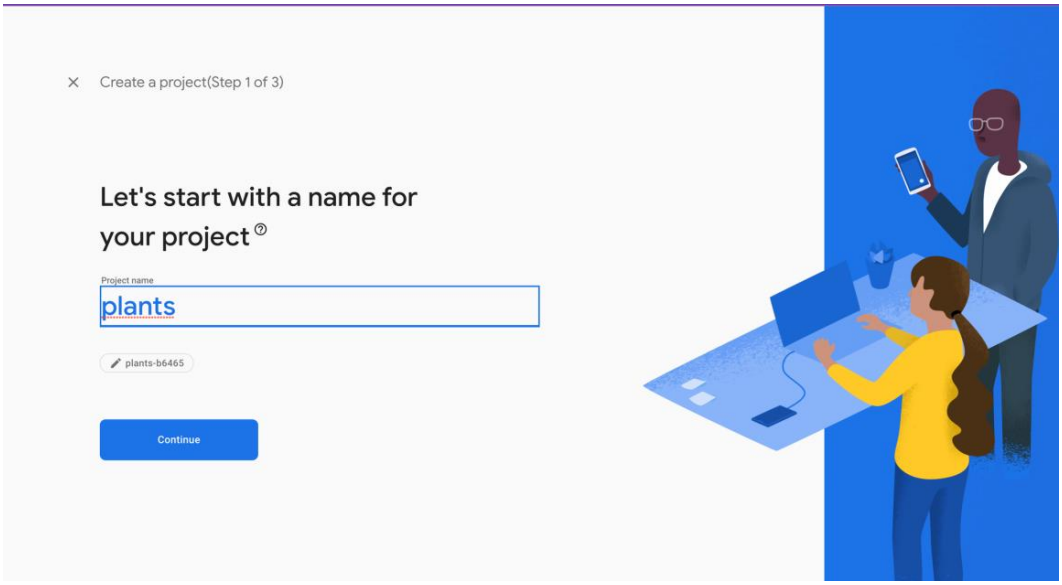

Figure 7 The project creation on Firebase

After that, I registered my Next.js application to Firebase. I also used the default option to add my application (Figure 8).

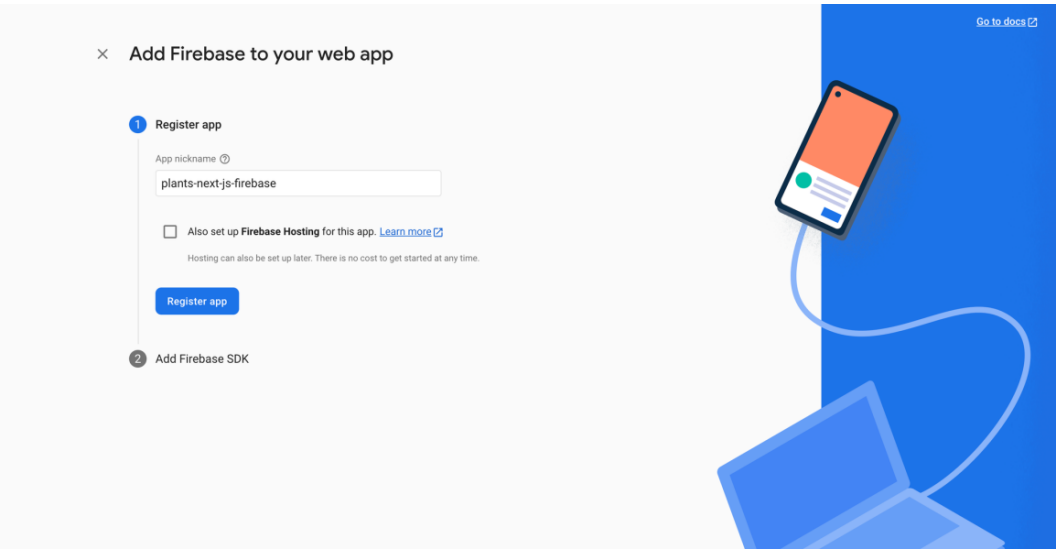

Figure 8 Adding Next.js application to Firebase.

To use Firebase in my Next.js application, I installed the Firebase SDK by using the commands:

Npm install firebase in Next.js project

npm install firebase @types/firebase

Next, I accessed the link ‘https://console.cloud.google.com/datastore/setup?project=plants-bf349’ to create a database for my Next.js application. Firstly, I selected the native mode (Figure 9 and Figure 10) since it offered enough functionality for my project. Then, I chose the ‘eu3’ (Europe) as my preferred location to store the data.

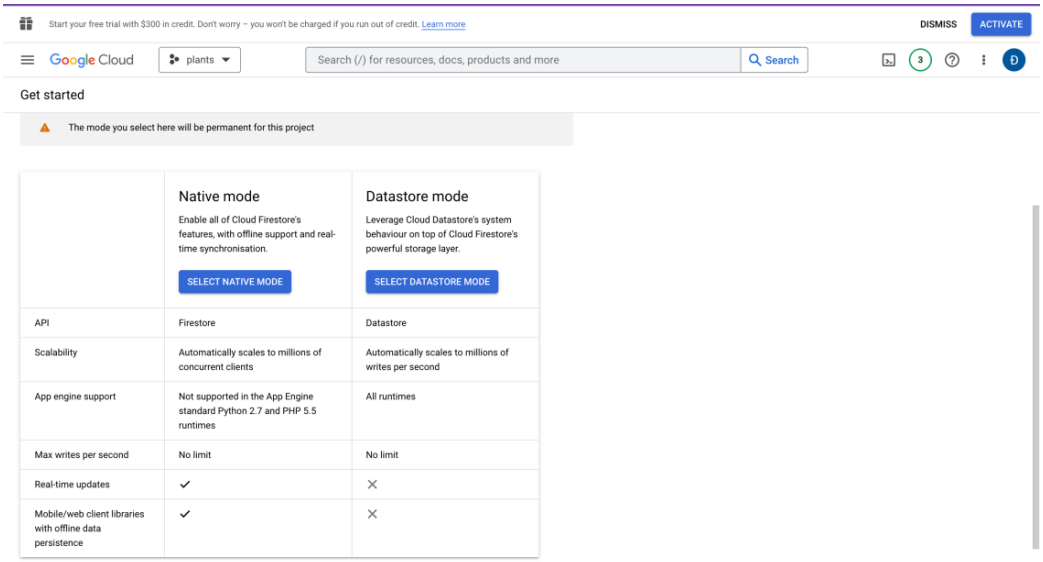

Figure 9 Creating a database with Native mode.

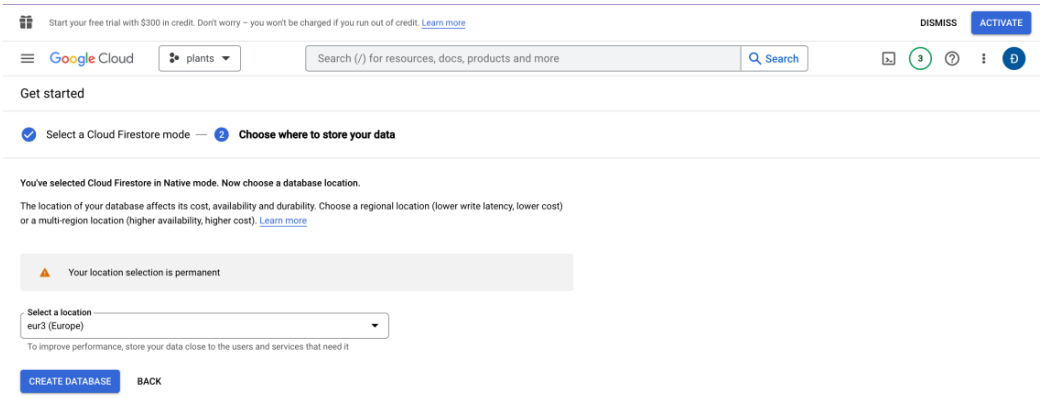

Figure 10 Configuration the database

As in the MongoDB database, I also created a collection to store the plan data. The name of the collection is Plants. I clicked the save button to initialize my collection (Figure 11 and Figure 12).

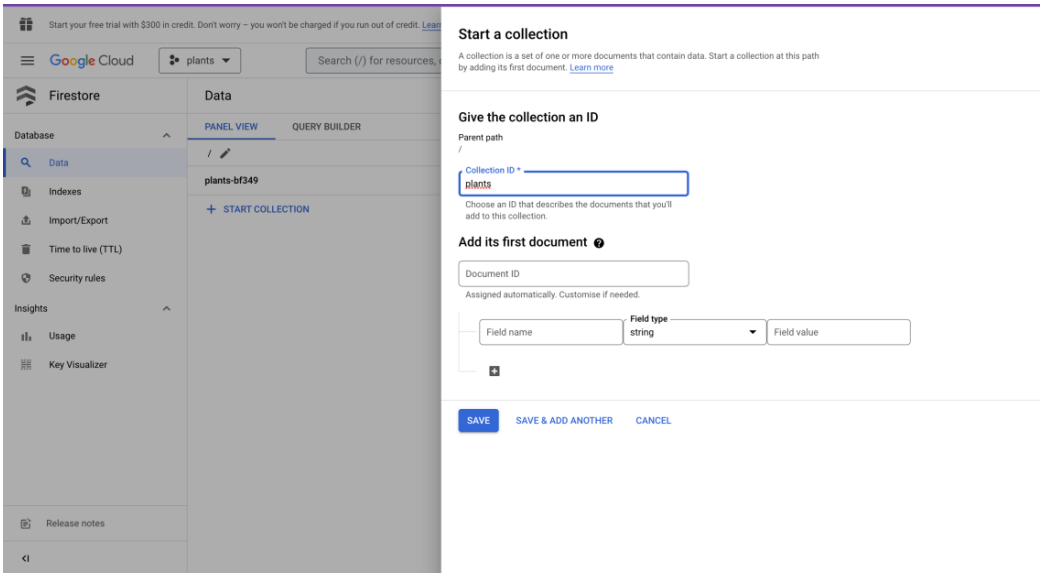

Figure 11 Create a collection in the database

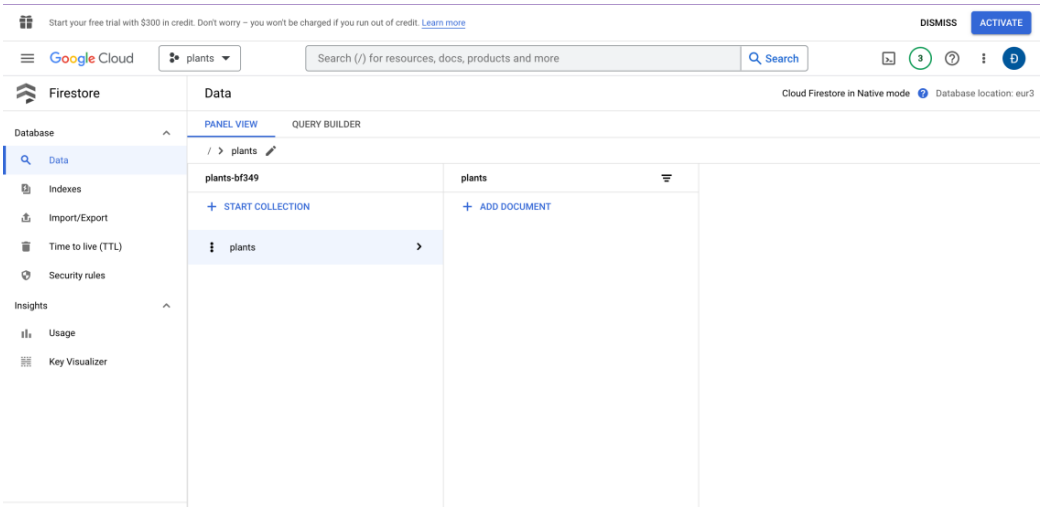

Figure 12 The collection of the database

In Firebase, I do not have permission to read or write in the database, so I clicked through the rules section to change the permission. Figure 13 is before editing and after I changed false to true, as shown in Figure 14, everyone accessing the database could read and write to the database.

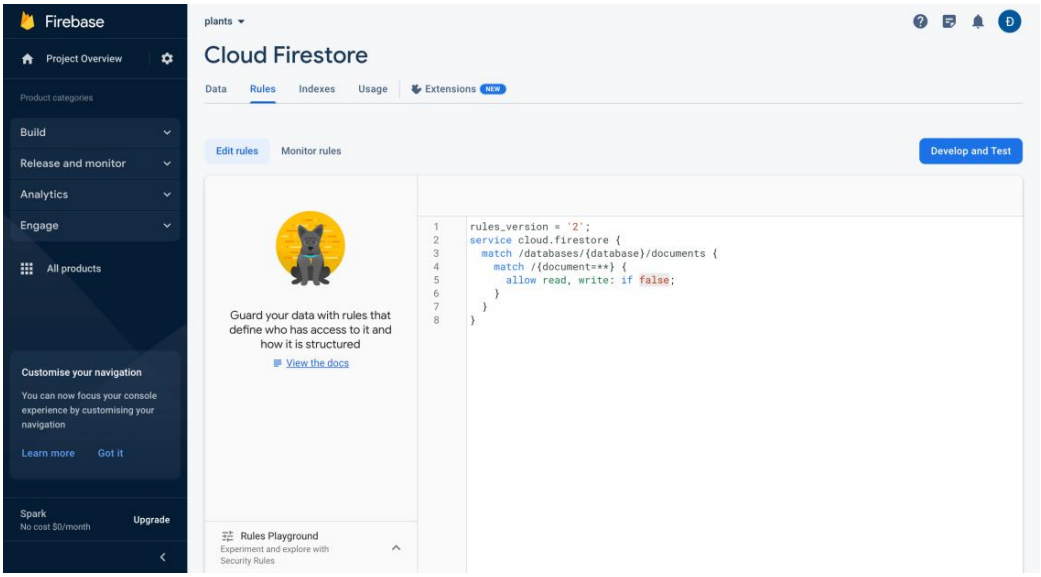

Figure 13 The Cloud Firestore's setting

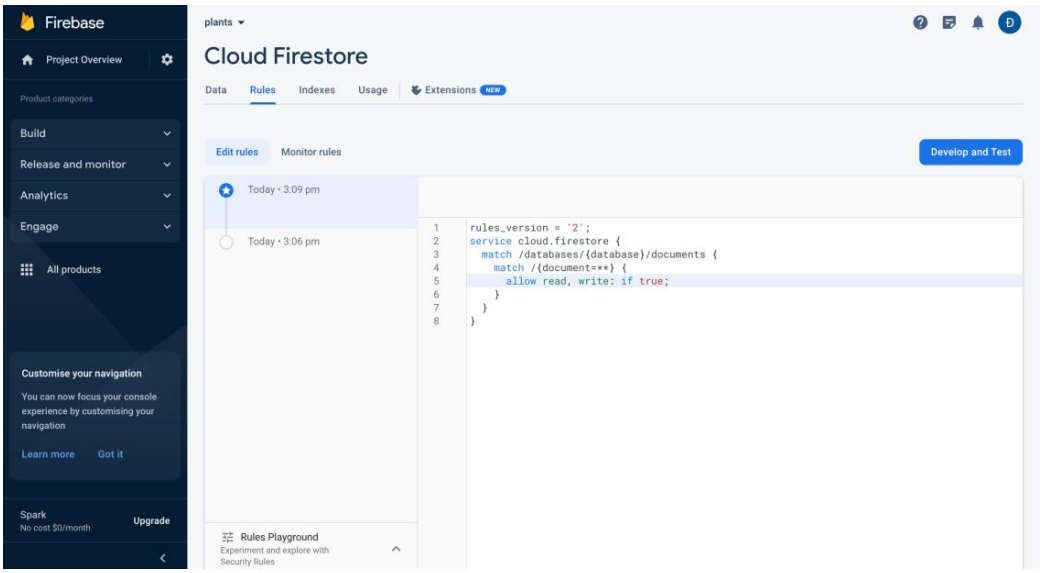

Figure 14 Changing the permission on the Cloud Firestore

After configuring all of that in the FireBase console, I configured my Next.js application. I created a new file in the root directory of your project called 'firebaseConfig.ts'. This file is used to initialize and configure the FireBase JavaScript SDK for the web application. First, I imported two functions which are the 'intializeApp' function from the 'firebase/app' module and the 'getFireStore' function from the 'firebase/firestore' module.

The ‘initializeApp’ function is used to initialize the FireBase app with the configuration object that includes API keys, authentication domain, project ID, storage bucket, messaging sender ID, app ID, and measurement (optional). The ‘getFirestore’ function is used to initialize FireStore. I took the ‘apikey’, which is generated from the FireBase project setting.

```
import { initializeApp } from 'firebase/app';
import { getFirestore } from 'firebase/firestore';

// Your web app's Firebase configuration
// For Firebase JS SDK v7.20.0 and later, measurementId is optional
const firebaseConfig = {
  apiKey: "MY_API_KEY",
  authDomain: "plants-bf349.firebaseapp.com",
  projectId: "plants-bf349",
  storageBucket: "plants-bf349.appspot.com",
  messagingSenderId: "693939373095",
  appId: "1:693939373095:web:524310a9ab4f852f20ba02",
  measurementId: "G-9MBHK52RCG"
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);

// Initialize Firestore
export const db = getFirestore(app);
```

Next, I also created an ‘index.tsx’ file in the root of my web application. The file was used to create a client-side application such as the frontend, and the function to populate data to the Firebase database. I first imported some necessary modules for my Next.js web application, including ‘API_BASE_URL, API_KEY’, which came from the ‘config.js’ file, and an API_KEY from the external API. I also imported ‘db’ from firebaseConfig to connect the module to the FireBase FireStore database. Additionally, I imported several moduls into this file:

```
import { API_BASE_URL, API_KEY } from '../../config';
import axios from 'axios';
import { db } from '../../firebaseConfig';
import { Inter } from 'next/font/google'
import { collection, query, where, getDocs, addDoc, deleteDoc, writeBatch } from 'firebase/firestore';
import { handleSearch } from '@/components/card';

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

Then, I used the updateAllPlants() function to populate the “plants” collection in the Firebase database with data from an external API. The function first deletes all existing documents in the collection and then retrieves plant data, which is stored in the plants array. Each plant object is added to the database using the addDoc() function. Once all the plant objects have been added to the database, a log message is printed to the console. If an error occurs during the database operation, the error is printed to the console.

The function is similar to the plantsCollection.deleteMany({}) function in Express.js.

```
async function updateAllPlants() {
 try {
   // Delete all documents in the "plants" collection
   const plantsCollectionRef = collection(db, 'plants');
   const plantsQuery = query(plantsCollectionRef);
   const snapshot = await getDocs(plantsQuery);
   const batch = writeBatch(db);
   snapshot.forEach((doc) => {
     batch.delete(doc.ref);
   });
   await batch.commit();
   // Make GET request to plant API and write all plants to database
   for (let i = 1; i <= 99; i++) {
     const response = await axios.get(`${API_BASE_URL}?page=${i}&key=${API_KEY}`);
     const plants = response.data.data;
     console.log(i)
     for (let j = 0; j < plants.length; j++) {
```

```
      const plant = plants[j];
      await addDoc(plantsCollectionRef, plant);
    }
   }

   console.log('Plants collection populated with data from API');

  } catch (err) {
   console.error(err);
  }
}
```

I created a function called Home() to render the frontend, which includes the button for fetching and writing data to the Firebase database from the external API.

```
export default function Home() {
 return (
  <main className={`flex min-h-screen flex-col items-center justify-between p-24 ${inter.className}`}>
   <h1 className="mb-4 text-4xl font-extrabold leading-none tracking-tight text-gray-900 md:text-5xl
lg:text-6xl dark:text-white">Plants <mark className="px-2 text-white bg-green-800 rounded dark:bg-green-
700">Wikipedia</mark></h1>
   <div className="flex items-center">
    <button className="bg-green-800 hover:bg-green-700 text-white font-bold py-2 px-4 rounded"
onClick={updateAllPlants}>
     Update all plants
    </button>
    <div className="relative ml-3">
     <div className="absolute top-3 left-3 items-center">
      <svg className="w-5 h-5 text-gray-500" fill="currentColor" viewBox="0 0 20 20"
xmlns="http://www.w3.org/2000/svg"><path fillRule="evenodd" d="M8 4a4 4 0 100 8 4 4 0 000-8zM2 8a6 6 0
1110.89 3.476l4.817 4.817a1 1 0 01-1.414 1.414l-4.816-4.816A6 6 0 012 8z"
clipRule="evenodd"></path></svg>
     </div>
     <input
      type="text"
      className="block p-2 pl-10 w-70 text-gray-900 bg-gray-50 rounded-lg border border-gray-300"
      placeholder="Search Here..."
      onKeyUp={handleSearch}
```

```
        />
      </div>
    </div>
    <div id="plantCards"></div>
  </main>
 )
}
```

After everything was configured, I ran the application by command npm run dev and clicked the “Updated all plants” button but I had an error presented in Figure 15. So, I clicked on the link it recommended.

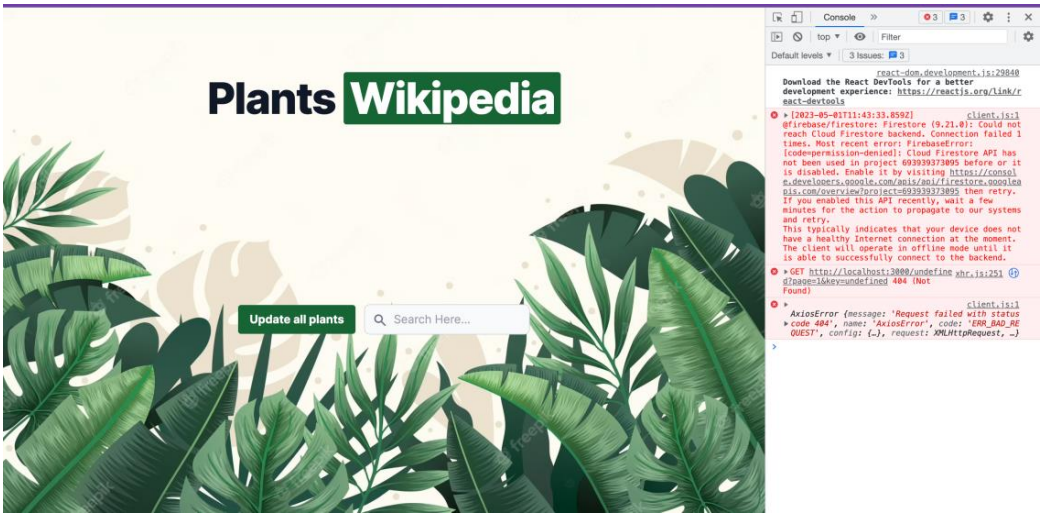

Figure 15 Error seen in browser console

Since I did not enable the FireStore API so the application cannot reach the Could FireStore backend. So, I enable it, presented in Figure 16.

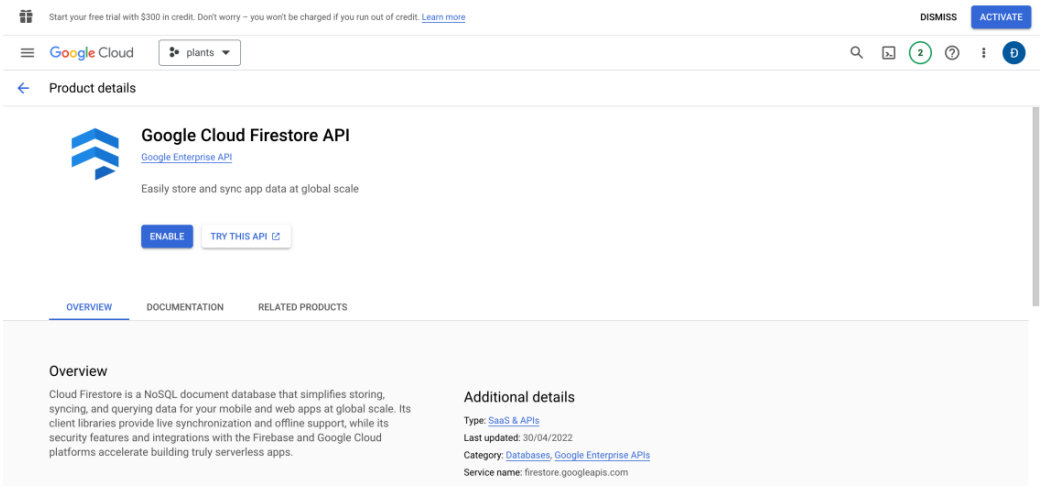

Figure 16 Enable the Google Firestore API

I clicked a button again and there was no error. Then I checked on the Firebase console and saw that there were data (Figure 17).

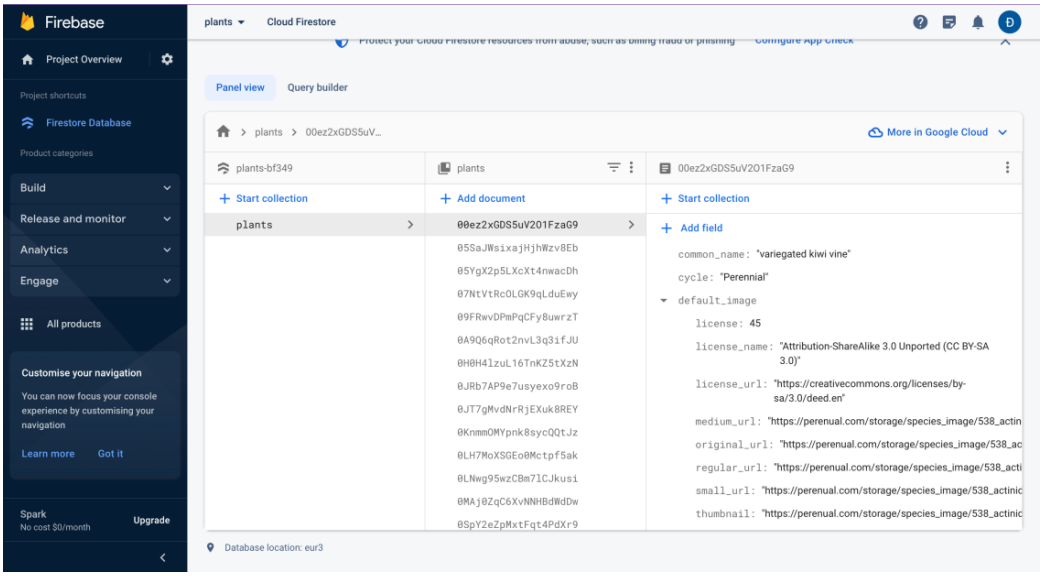

Figure 17 The data in the database

I created the 'card.tsx' file to render the plant data information. The function is similar to the 'card.tsx' file in the Express.js project, but it uses different functions. I imported the 'axios' module to make an HTTP GET request to an external API and retrieve the plant data, which I then updated in the Firebase Firestore database. Additionally, I imported the 'Image' module from 'next/image/' to get the

image description from the domain and assign it to the card component. Some of the modules used in this file are similar to the ones used in the 'index.tsx' file.

```
import React from 'react';
import axios from 'axios';
import Image from 'next/image';
import ReactDOM from 'react-dom';
import { db } from '../../firebaseConfig';
import { query, where, orderBy, collection, getDocs } from 'firebase/firestore';
```

After that, I adapted code similar to what I used in the Express.js project to render the card component. I made some modifications to make it work with the Firebase console.

```
interface PlantData {
   id: number;
   common_name: string;
   scientific_name: string[];
   other_name?: string[];
   default_image: {
    original_url: string;
    small_url: string;
    medium_url: string;
    thumbnail: string;
    license: number;
    license_name: string;
    license_url: string;
   };
   sunlight: string[];
   watering: string;
   cycle: string;
 }

interface CardProps {
  plantData: PlantData;
}

const Card: React.FC<CardProps> = ({ plantData }) => {
  return (
```

```
    <div className="card-container">
      <div className="max-w-sm rounded overflow-hidden shadow-lg bg-white">
        <Image width={500} height={500} className="w-full" src={plantData.default_image?.medium_url}
alt={plantData.common_name} />
        <div className="card-body px-6 py-4">
          <div className="font-bold text-xl mb-2">{plantData.common_name}</div>
          <p className="text-gray-700 text-base">Scientific name: {plantData.scientific_name}</p>
          <p className="text-gray-700 text-base">Cycle: {plantData.cycle}</p>

        </div>
        <div className="card-body px-6 py-4">
          <div white-space={"nowrap"}>
            <span className="inline-block bg-gray-200 rounded-full px-3 py-1 text-sm font-semibold text-
gray-700 mr-2 mb-2">
              Sunlight: #{plantData.sunlight}
            </span>
          </div>
          <div white-space={"nowrap"}>
            <span className="inline-block bg-gray-200 rounded-full px-3 py-1 text-sm font-semibold text-
gray-700 mr-2 mb-2">
              Watering: #{plantData.watering}
            </span>
          </div>
        </div>
      </div>
      <br></br>
    </div>
  );
};
```

I used the function handleSearch() to search for plants in the Firebase FireStore database. It is called when the user type in a search query and presses the Enter key. The function extracts the search query from the input field, converts it to lowercase, and removes extra spaces. It then uses the search query to filter the plants in the database by their “comman_name” field using the ’where()’ method. The results of the search are displayed as a list of cards using the ‘ReactDOM.render()’ method. The handleSearch()’ function allows users to

search for the plants by their common name and view the results in a visually appealing format.

```
export async function handleSearch(event: any) {
  if (event.keyCode === 13) {
    // The Enter key was pressed, so get the plant name from the input field
    const plantName = event.target.value.toLowerCase().trim();
    // perform the search using the plantName
    try {
      const plantsCollectionRef = collection(db, 'plants');
      const q = query(plantsCollectionRef,
        where('common_name', '>=', plantName),
        where('common_name', '<=', plantName + '\uf8ff')
      );
      const snapshot = await getDocs(q);

      const plantDataList: PlantData[] = [];

      snapshot.forEach((doc) => {
        const plantData = doc.data() as PlantData;;
        plantDataList.push(plantData);
      });

      const cards = plantDataList.map((plantData) => (
        <Card key={plantData.id} plantData={plantData} />
      ));

      ReactDOM.render(<div>{cards}</div>, document.getElementById('plantCards'));
    } catch (error) {
      console.error(error);
    }

  }
}
```

Figure 18 presents the final Next.js application with Firebase. The search engine was able to find multiple types of plants when the keyword was entered into the search bar.

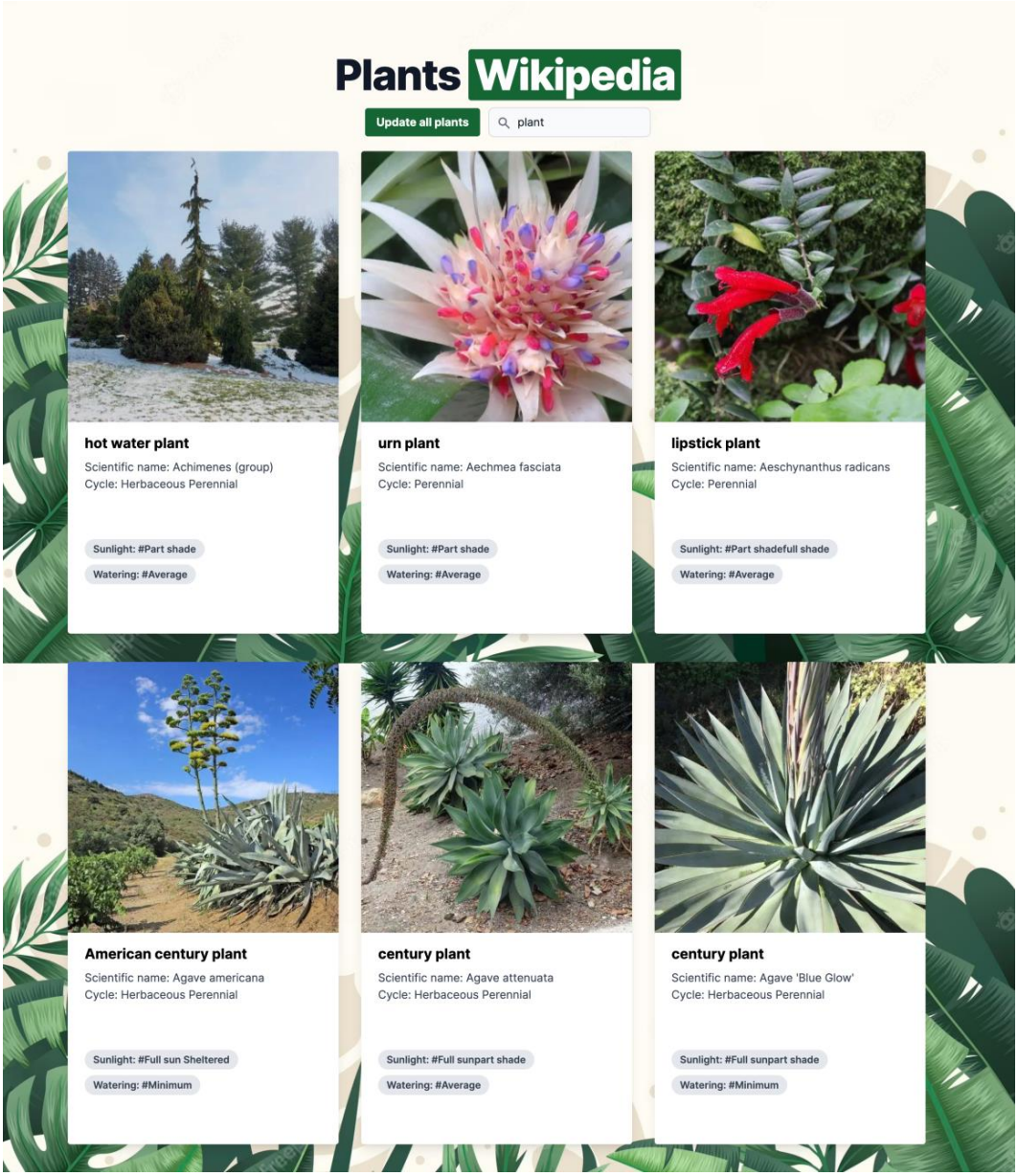

Figure 18 The final Next.js application with Firebase

Figure 18 represents the result of the plant information using Firebase. Details are the same as Express.js with MongoDB.

3.2.3. Comparison and conclusion

About functionalities, although my scenario application was fairly simple, I did manage to test some basic but necessary features a backend solution must

provide, such as providing APIs and reading/writing to the database. Both solutions worked well with my Next.js frontend, both served the purpose with little different syntax. To compare their functionalities on a higher level, they offer a range of different features and capabilities.

Table 1 Comparison between Express.js with MongoDB and Firebase

	Express.js with MongoDB	**Firebase**
Backend Architecture	The developer completely controls the backend architecture, allowing to build APIs, Routes, and middleware.	Firebase provides pre-built backend services and tools.
Database	MongoDB is document-oriented database that provides a flexible schema and advanced querying capabilities.	Firebase provides a real-time database that is optimized for real-time data synchronization.
User Authentication	Does not provide built-in user authentication	Provide a built-in user authentication system
Hosting	Does not provide a built-in hosting service	Provide a hosting service
Analytics and Monitoring	Does not provide built-in analytics and monitoring tools, but there are third party services and tool that can be used for this	Provide built-in analytics and monitoring tools
Security	More secure	Less secure
Suitable	Large-scale application. The sharding and replication features of MongoDB allow it to scale horizontally to handle huge volumes of data.	Small-scale application. The ideal for applications that require real-time data synchronization between clients.
Performance	MongoDB offers a significant performance advantage. The servers are stable	Firebase stores data on a cloud platform that makes the server unreliable and restricts its flexibility.
Cost	MongoDB is free to use	Firebase can be more expensive than other options, particulary for large or complex applications.

Table 2 Pros and cons of Express.js with MongoDB and Firebase

	Express.js with MongoDB	**Firebase**
Pros	Excellent for developing RESTful APIs and web applications. MongoDB offers great scalability and performance. Allows for flexible data modeling with MongoDB's document-based data model.	Firebase is highly scalable and can handle large volumes of data traffic. Firebase offers seamless integration with other Google services. Firebase offers great security features and has a strong focus on protecting user data.
Cons	MongodB's query language can be difficult to learn and understand. Security can be a concern if not properly configured. Scaling becomes more difficult as the database grows in size and complexity.	Firebase can be limiting in terms of data modeling and querying, as it uses a NoSQL data model. Firebase has limited support for complex transactions and joins.

Regarding ease of use, I faced some challenges while installing and connecting to MongoDB on my MacOS computer, as there were issues related to the root directory being unavailable for external applications. When attempting to connect to the server database, I encountered a 404 error, which was difficult and time-consuming to debug. However, once I overcame these initial difficulties and managed to install MongoDB, creating a collection became a much simpler task. In contrast, installing Firebase was a simpler process. It involved creating a Firebase project on the Firebase website and configuring my app to use Firebase services. Additionally, Firebase provided helpful error messages that pointed out the issues I was having, and I was able to quickly resolve them by following their instructions.

Regarding documentation and community, the official Express.js documentation is well-organized and easy to navigate, with clear explanations and examples for each component of the framework. MongoDB documentation is also thorough and includes a variety of resources such as tutorials and manuals, as well as API references. Express.js and MongoDB are one of the most active and largest communities, It has an active GitHub presence and Forums, where I can post issues and find the answers easily. On the other hand, Firebase also has an

official document from Google, which covers a variety of topics, from getting started to advanced features. The document is well-organized and includes detailed guides and examples, as well as API. Although Firebase was born after express.js with MongoDB, its community is also very large and active. Firebase also has a wide range of resources available including forums, user groups, and live courses to help newbies get started with Firebase. Firebase is also on GitHub, where users can submit problems and find solutions.

In addition to Express.js with MongoDB and Firebase, there are still many backend solutions that can be integrated with Nextj.js:

- Strapi is a headless CMS that can be used to create APIs and manage content. Strapi's REST and GraphQL APIs can be used to combine it with Next.js.
- Prisma - Prisma is an open-source object-relational mapping (ORM) that can be used to facilitate database access and maintenance. Prisma's GraphQL API allows it to be integrated with Next.js.
- AWS Amplify - AWS Amplify is a cloud-based backend solution that offers a collection of pre-built services and capabilities such as user authentication, data storage, and API management.

4 CONCLUSION

Next.js is a relatively new framework, which means that many developers are currently using it to build their applications. Therefore, the goal of this thesis is to compare different backend solutions such as Express.js with MongoDB and Firebase, based on a Next.js application. The topic is relevant for any developer who is looking for a suitable backend solution for their project. Although my major is in frontend development, specifically designing web application interfaces, I was able to learn a lot about backend development through this thesis, including how to create a database or collection, how to fetch data from an external API and store it in a database, and how to integrate it into a Next.js application. The background theory section explains the concepts and definitions related to the technologies and software development required to implement the practical

aspects of the thesis. The implementation section describes the process of creating the Next.js application and database, as well as how to connect the application to the database server. The application achieved the predefined goals set in the beginning. After testing the two solutions, I found that both worked well with my application. Although I faced some difficulties while installing MongoDB, I was able to resolve them easily by searching for solutions on the internet. I compared only two backend solutions in this thesis, but there are many other options available that can be integrated with Next.js, allowing developers to select the solution that best suits their needs.

REFERENCES

Jazayeri, M. 2007. Some Trends in Web Application Development. In: *Future of Software Engineering (FOSE '07)*. May 2007. 199–213. [Online]. Available at: doi:10.1109/FOSE.2007.26. [Accessed 7 April 2023].

Sasikumar, S., Prabha, S. & Mohan, C. 2022. *Improving Performance Of Next.js App And Testing It While Building A Badminton Based Web App*. [Online]. Available at: doi:10.2139/ssrn.4121058 [Accessed 18 April 2023].

Sianandar, J. & Manuaba, I. B. K. 2022. Performance Analysis of Hooks Functionality in React and Vue Frameworks. In: *2022 International Conference on Information Management and Technology (ICIMTech)*. August 2022. pp.139–143. [Online]. Available at: doi:10.1109/ICIMTech55957.2022.9915183. [Accessed 19 April 2023].

Express - Node.js web application framework. [Online]. Available at: https://expressjs.com/ [Accessed 29 April 2023].

Firebase. [Online]. Available at: https://firebase.google.com/ [Accessed 7 April 2023].

MongoDB Atlas Database | Multi-Cloud Database Service. [Online]. Available at: https://www.mongodb.com/atlas/database [Accessed 6 April 2023].

Perenual | Free Plant API | Houseplants,Garden,Trees,Flowers,Images & Data. [Online]. Available at: https://perenual.com/docs/api [Accessed 6 April 2023].

Quick Start – React. [Online]. Available at: https://react.dev/learn [Accessed 5 April 2023].

Routing: Pages and Layouts | Next.js. [Online]. Available at: https://nextjs.org/docs/pages/building-your-application/routing/pages-and-layouts [Accessed 5 April 2023].