OULUN AMMATTIKORKEAKOULU

Huy Truong

BUILD AND DEPLOY A HIGH-PERFORMANCE FULL STACK JAVASCRIPT WEB APPLICATION

BUILD AND DEPLOY A HIGH-PERFORMANCE FULL STACK JAVASCRIPT WEB APPLICATION

Huy Truong
Bachelor's thesis
Spring 2020
Information technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree programme in Information Technology

Author: Huy Truong
Title of the thesis: Build and Deply a High-performance full stack JavaScript Web Application
Thesis examiner: Timo Vainio
Term and year of thesis completion: Spring 2020 Pages: 49

Web development has been developed rapidly in recent years, especially regarding to web development with JavaScript. JavaScript web development has become the new standard in the tech industry with most job is required to know React, Node.js which is developed in JavaScript. React, an open-source project developed by Facebook which has proven to be one of the fastest rendering libraries, has a variety of services and a huge community to support it.

This thesis aimed to build and deploy a discussion forum web-application using only JavaScript. This thesis provides a brief explanations and motivation for each of the libraries used in the projects. The application is a social platform where users can make a post about a certain topic and other users can rate it as well as make comments and interact with each other. The frontend was created with React (Next.js) with server-side rendering, Apollo as a state management that manages local and remote date with GraphQL. Nodejs with Express are used for back-end with PostgreSQL as a database, GraphQL is a query language used for the API to retrieve data.

Keywords: React, Next.js, Node.js, PostgreSQL, GraphQL, Express, server-side rendering, API, libraries, web development

CONTENT

VOCABULARY

API: Application program interface

CSS: Cascading Style Sheets

DOM: Document Object Model

DDos: Denial of Services

HTML: Hypertext Markup Language

I/O: Input/Output

JSON: JavaScript Object Notation

NPM: Node Package Manager

ORM: Object–relational mapping

SEO: Search Engine Optimization

SSH: Secure shell

URL: Uniform Resource Locator

VDOM: Virtual Document Object Model

XML: Extensible Markup Language

1 INTRODUCTION

Web development has always been there since the beginning of the Internet, where the first website was created in 1991. Web technology has developed significantly from a few static pages combine together using plain HTML and CSS. Now a website has become larger, more complex and it can become an industry as its own. A large amount of framework and libraries has been developed to keep it up with the demand for light weight, high performances, SEO friendly and scalability.

JavaScript has risen to become the most popular language for web development, as it is the language that applies for both frontend and backend. Many libraries and framework have been developed over the years, with only some stand out for the utilities they provide. React is a fast, scalable and simple library used for building user interfaces for a single page application. Express.js is a minimal and flexible Javascript server application framework, it is scalable and has a huge community. Graphql has become a new revolutionary way about an API, allowing the user to fetch all the necessary data within one fetch.

This thesis demonstrates the development of a discussion forum web application using the full-stack JavaScript technology. The start of the thesis will discuss in dept on the stack technologies, compatibilities and advantages of all the technology involved in the application.

2 TECH STACK TECHNOLOGY

2.1 React

React is an open-source JavaScript library developed by Facebook, used for building user interfaces for single page application.

2.1.1 JSX

JSX is an extension to JavaScript that allows the XML/HTML syntax use alongside with JavaScript. JSX does not work on a browser, so it has to be converted into JavaScript for the browser to understand by using a tool such as Babel, it will scan over the syntax for the XML/HTML syntax and use creatElement() to create the element shown in figure 1.(1)

```
js
    <h1>Yay!</h1>

    Into this:

js
    React.createElement('h1', null, 'Yay!')
```

Figure 1. An example of JSX convert into Javascript

2.1.2 Declarative (DOM and VDOM)

React implemented with VDOM is declarative, allowing it to efficiently change data, render and update the declared components and remove the inefficiency of normal DOM. A website is composed of DOM (Document Object Mode), an internal programmatic in the browser that represents the webpage. It is used to manipulate HTML and XML documents structure, style, and content

without needing to go to the document itself. (2). DOM has a tree structure shown in figure 2 which makes it easy to traverse but it can be very slow in performance. Most of modern web applications are dynamic SPA (Single page application), having huge DOM trees and traversing through them and making a change can be difficult to manage and inefficient. VDOM is an extremely fast and light weight abstraction of the browser DOM in a JSON object. Whenever a change is made, a VDOM is made to compare to the browser DOM. Then it will create a list of changes and apply them all in a single render. (3)

```
<Table>
   <ROWS>
      <TR>
         <TD>Car</TD>
         <TD>Scooter</TD>
      </TR>
      <TR>
         <TD>MotorBike</TD>
         <TD>Bus</TD>
      </TR>
   </ROWS>
</Table>
```

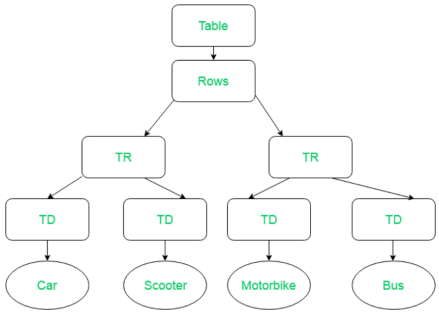

Figure 2. An example of DOM trees structure

2.1.3 Component-based

React is core strength is in its component-based structure. Components are parts of the user interfaces. They can be reused and nested in other components. There are two types of components, functional components and class components. Functional components are just a JavaScript function that can optionally accept props and return HTML (JSX). A class Component is an ES6 class that is extended from React libraries, having private data, such as a state, lifecycle hooks and a need render method to return HTML. Each component has its own internal logic and can be reuse across the application. It helps large web applications code structure more consistent, easy to scale and manage (4). As shown in figure 3, there are 2 types of components in React which are class and functional components.

```
import React from 'react';
import PropTypes from 'prop-types';

class Hello extends React.Component {
    render() {
        const {greeting, firstName} = this.props;
        return (
            <div>
                {greeting} {firstName}
            </div>
        )
    }
}

export default Hello;
```

```
import React from 'react';
import PropTypes from 'prop-types';

function Hello({greeting, firstName}) {
    return (
        <div>
            {greeting} {firstName}
        </div>
    )
}

export default Hello;
```

Figure 3. class and functional components example (5)

2.1.4 React Hooks

A functional component is a stateless component it does not provide a state or lifecycle unlike class components. React Hook solves this problem allowing it to hook the state and lifecycle feature into function components. Functional components are preferred during development for their simplicity, less code and easier to read. With React Hook, a functional and class component are almost identical to one another. (6).

2.2 Next.js (Server-side rendering)

Next.js is a wrapper built around the React framework, enabling React to pre-render their code on the server. With React, Next.js provides client-side rendering which enables their website to render entirely on JavaScript, JavaScript bundle code is downloaded from the server and then that code is executed on the browser and it shows the output HTML element. Next.js provides server-side rendering allows client-side framework to render static HTML and CSS from the server shown in figure 5. The static HTML and CSS will appear on the source page, improved performances and SEO for better search rate (7). Unlike client-side rendering shown in figure 4, HTML and CSS are rendered by JavaScript bundle downloaded from the server.

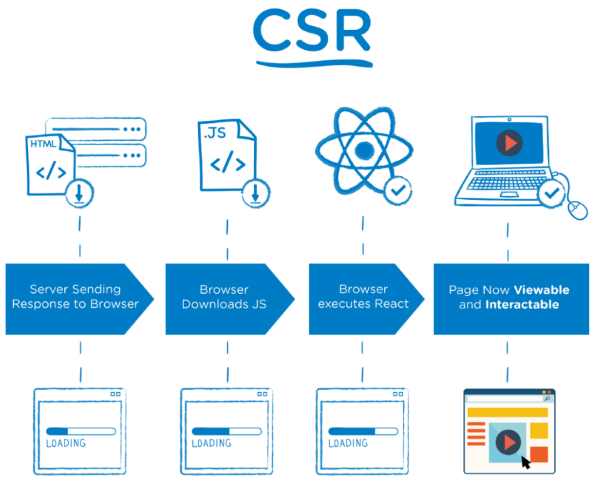

Figure 4. client side

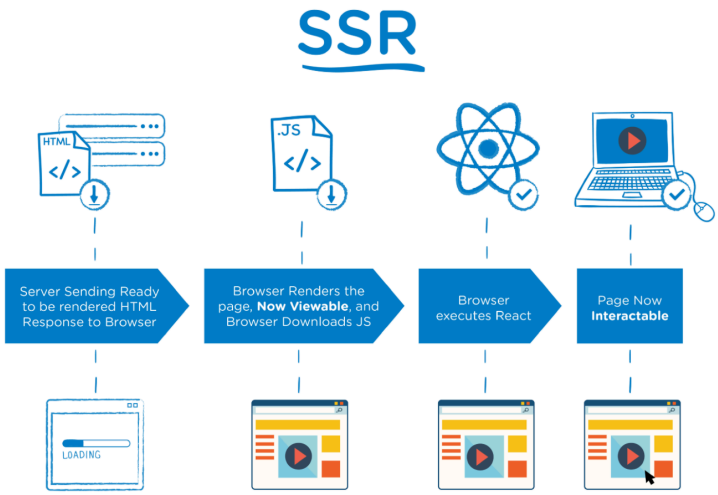

Figure 5. server side

2.3 Node

Node is a runtime environment based on Chrome's V8 engine for executing JavaScript outside the browser, commonly used to build back-end services. Node offers an event-driven non-blocking I/O and an asynchronous request handling capable of handling multiple requests without delays, enabling the developer to use real-time applications (8). Node has a high performance and it is light weight, perfect for asynchronous programming and provides a high scalability. Node has a huge amount of available packages in NPM and a large community for supports. Node provides advantages to write server-side with JavaScript. With its feature improved and supported by the community over the years, it is now widely used by large companies such as Netflix, Uber, PayPal.

2.4 Express

Express is a prebuilt Node routing and middle framework that allows the developer to create and maintain a server. As shown in figure 6, Express middleware are functions that are executed during a request lifecycle, it has access to the HTTP request object (req) and response object (res) for each route it is attached to. The middleware function follows the request-response life cycle. If the cycle is not complete, it will call the next() function, a function to pass control to another middleware function and enable it to handle a request asynchronously.

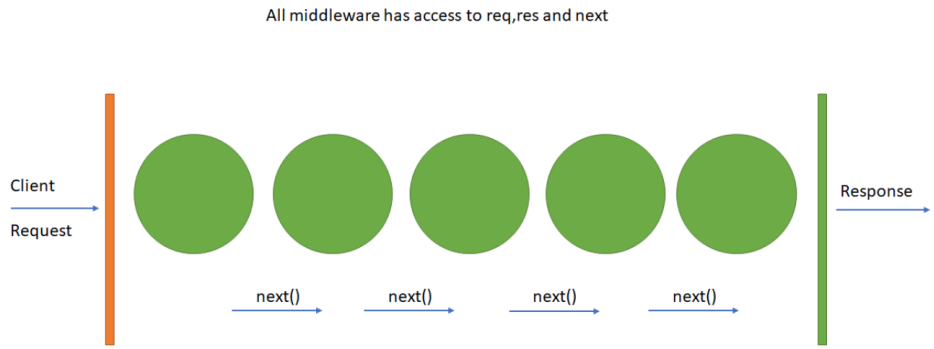

Figure 6. Express request-response lifecycle (9)

2.5 GRAPHQL

GraphQL is a query language that receives data from the server into applications developed by Facebook. Usually, with a REST API, the developer needs to make a route handler and provide a different endpoint for different function while GraphQL provides a function that matches a particular field on a type. For example, a Query type is made for a resolver that returns a “me” field value. Then it is written as shown in figure 7. When performing a GraphQL query, it will fetch the requested data that matches the shape of the query back from the API instead of the whole dataset, like the REST API. On the client, GraphQL is declared in a simple, easy to understand structure to fetch the required data

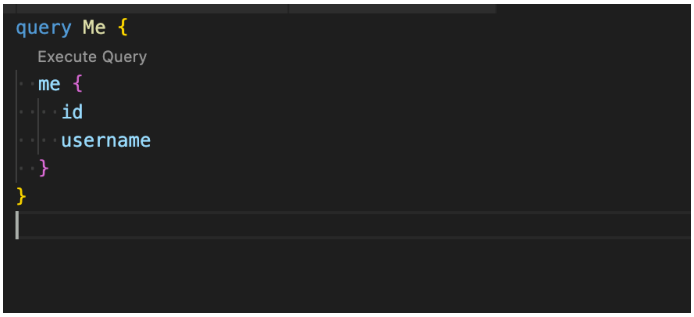

Figure 7. Graphql queries example

On the server, GraphQL involves a schema and a resolver function. The Schema is a model for the data. It determines what data the client can fetch and the types of the data.

In relational database, REST need to make multiple queries to fetch data references to one another. As shown in figure 8, The Graphql schema describe how each other data are related and data can be collected from multiple sources in a single query, saving bandwidth and reduce requests.

```
@ObjectType()
@Entity()
export class User extends BaseEntity {
  @Field()
  @PrimaryGeneratedColumn()
  id!: number;

  @OneToMany(() => Post, (post) => post.creator)
  posts: Post[];

  @OneToMany(() => Updoot, (updoot) => updoot.post)
  updoots: Updoot[];

  @OneToMany(() => Comment, (comment) => comment.user)
  comments: Comment[];

  @Field()
  @Column({ unique: true })
  username!: string;

  @Field()
  @Column({ unique: true })
  email!: string;

  @Column()
  password!: string;

  @Field(() => String)
  @CreateDateColumn()
  createdAt: Date;
```

Figure 8. Graphql types and fields example

2.6 PostgreSQL

PostgreSQL is a powerful, open-source object-relational database management system, a database that forms a relation between tables that store data on specific entities. A relational database provides simplicity, data accuracy, data integrity, flexibility and security. PostgreSQL do not belong to any company but is supported by a huge community of developers around the globe. PostgreSQL uses SQL, a computer language for storing, manipulating and storing data in a relational database. PostgreSQL applies the client-server model. The client and the server can be on different hosts in the network environment. As shown in figure 9, the server connected to the database accepts a request from the client application, execute the request and sends the result back to the client. (10)

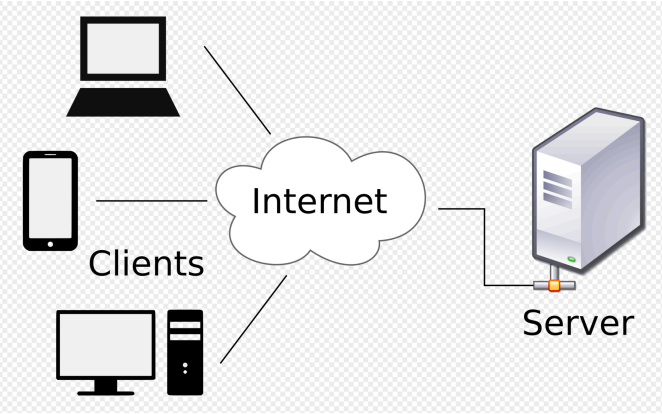

Figure 9. Client connect with server through internet

2.7 ORM

An Object-Relational Mapping (ORM) is a technique that allows the developer to query and change data from the database in an object-oriented way. An ORM is usually referred to a library that uses the Object-Relational Mapping technique so that the developer does not have to write a raw SQL to manipulate data.

2.8 Redis

Redis is one of the most popular NoSQL databases. It is a popular open-source in-memory data structure used for cache on the database level. It is written in C, run entirely in memory and it is

optimized to deliver millions of operations per second with a millisecond of latency within a standard server. Redis has pre-built popular use-case for a data structure, such as bitmaps, hyperloglogs, geospatial, indexes strings, sorted sets with range queries, hashes, lists, sets and streams. Redis addresses complex functionality with simple a command, enables data to run on the database level and not on the application. This reduces the code complexity and bandwidth requirement. Redis can be flexible with different data use cases with Redis modules, allowing the user to customize functionality to adapt data according to database (11). As shown in figure 10, Redis store program functions and data that used repeatedly in Redis cache and reused them across the application.

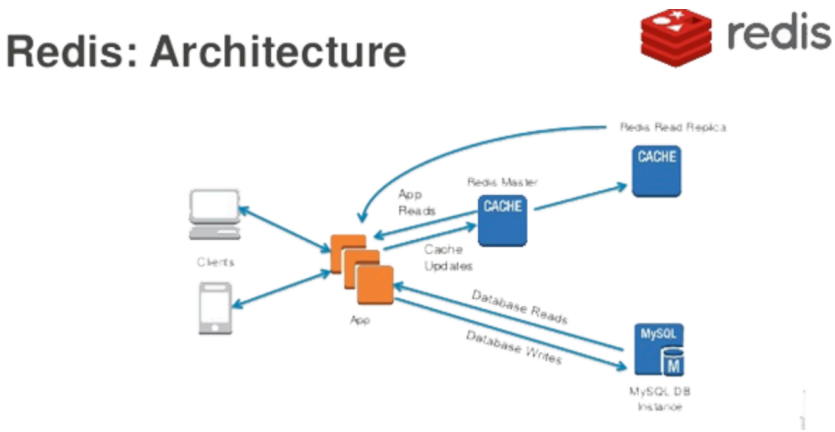

Figure 10. Redis workflow

2.9 TypeScript

TypeScript is an open-source language built on JavaScript developed by Microsoft. It provides type checking for types, classes and interfaces. TypeScript ensures that a function takes in and produce the correct value and ensure that an application scaling all the functionality will work as intended. TypeScript can easily convert to JavaScript with a TypeScript compiler or Babel. It can be use where JavaScript code runs. TypeScript helps the developer to better scale their application, reduce and handle an unexpected behaviour and bugs.

2.10 Docker

Docker allows the developer to package and run the application in a container, an isolated environment which is highly compatible and able to run on every system. Docker has all the functionality of a virtual machine. While being more lightweight, it does not occupy a lot of memory space, good boot up time, high performance, high scalability and efficiency.

3 PROJECT REQUIREMENT AND SETUP

This thesis project aimed to create a discussion forum based on “Reddit”, a popular social media that people can rate each other’s posts and opinions on anything. This section indicates the project requirement and the implementation of the project.

3.1 Project requirement

This project will contain the basic functionality of the website “Reddit”. When a user visits the web-site, they are able to see all the content of the page. All of the other user posts and comments will display for all users and they can use the search bar to search for the content they are interested. The forum has an authorization form for a user to log in, since some of the functions are limited to a logged in user such as creating post, upvoting and downvoting a post, commenting on a post, editing and deleting their own post. If the user has not logged in and tries to use these functions, they will be redirected to the login form where they can log into their account.

3.2 Development environment setup

A local development environment needs to be set up before implementing the application for a smooth development process.

3.2.1 Text editor (Visual Studio code)

Visual Studio code is one of the most popular free open-source text editors. It can be used in Mac, Windows and Linus. It has a built-in feature to support Node and JavaScript. Other technologies can be supported through an extension. There are a lot of extensions to help in Visual Studio code to support the development process, such as auto styling, debugger, style checking. (12)

3.2.2 Version control (Git)

Git is one of the most popular version controls. It keeps track of all the code the user makes and it can revert back and forth through a different version of the application. Git is very helpful in team collaboration. It allows different users to make a change to the code at a different branch and merge all the change into one source. As shown in figure 11 below, the “master” branch is the official release while the “develop” branch is the branch the user uses for development. Then it can be merged into a “master” branch for another release. (13)

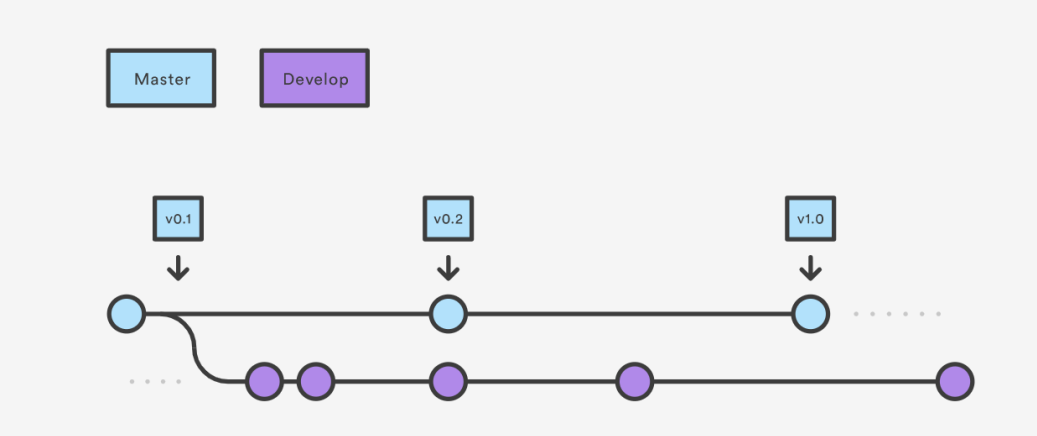

Figure 11. Git develop and master branch (10)

3.2.3 Server Environment and Package Manager (Node and Yarn)

Node is used as a server environment and it needs to be installed locally before implementing the application. NPM comes with Node as a package manager, but Yarn is used for this project instead since it installs libraries faster. (14)

4 IMPLEMENTATION

The development process of the discussion forum will be split into three parts: the backend, frontend and the deployment of the applications. This section is dedicated to explain the development of all the steps in detail.

4.1 Backend implementation

The backend development of the discussion forum included the initial set up, connection to the database and creation of Entity Resolver to manipulate the database.

4.1.1 Backend set up

To the start of backend development, run command in figure 12 in the terminal to generate a package.json file. A package.json is a JSON file that keeps track of any change related to the dependencies, author, license, scripts and version of the projects.

```
Apples-MacBook-Pro:thesis bachelor apple$ npm init
```

Figure 12. Command to initialize the projects backend

Tsconfig.json can be created by running the command in figure 13. The file allows the developer to configure the Typescript compiler options to compile any Typescript file for Node.

```
Apples-MacBook-Pro:thesis bachelor apple$ run tsc --init
```

Figure 13. Command to create Typescript compiler configuration file

After that a main file src/index.ts is created. It will be the main entry for the backend application and it is a place to setup typeORM, starting with connecting to a database. An empty PostgreSQL database can be created with a postgres application. Then it is connected to the database by using a typeORM function called “createConnection”. The connection requires entities to be defined which will be mentioned in the later section. As shown in figure 14, typeORM has a function that connects the server to the postgreSQL database.

```
await createConnection({
    type: "postgres",
    url: process.env.DATABASE_URL,
    logging: true,
    // synchronize: true,
    migrations: [path.join(__dirname, "./migrations/*")],
    entities: [Post, User, Updoot, Comment],
});
```

Figure 14. typeORM functions to connect database to backend application

As shown in figure 15 below, Express is imported from an “express” module to help start the app and listen to port define in a “dotenv-safe” package where it defines all the applications secrets such as token, port, session secret in an “.env” file. Then in figure 16, a redis server is established to stored data in session. Secure site and data expiration date are also defined here.

```
import "dotenv-safe/config";
import express from "express";
import path from "path";
import { createConnection } from "typeorm";
import { Post } from "./entities/Post";
import { Updoot } from "./entities/Updoot";
import { User } from "./entities/User";

//reru
const main = async () => {
  const conn = await createConnecti
    type: "postgres",
    url: process.env.DATABASE_URL,
    logging: true,
    // synchronize: true,
    migrations: [path.join(__dirnam
    entities: [Post, User, Updoot, Comment],
  });

  const app = express();
  app.listen(parseInt(process.env.PORT), () => {
    console.log(`server started on ${process.env.PORT}`);
  });
};

main().catch((err) => {
  console.error(err);
});
```

```
var Comment: {
    new (data?: string | u
    prototype: Comment;
}

Textual notations within markup; a
comments are available to be reac
```

Figure 15. Content inside index.js file

```
const RedisStore = connectRedis(session);
const redis = new Redis(process.env.REDIS_URL);
app.use(
  session({
    name: COOKIE_NAME,
    store: new RedisStore({
      client: redis,
      disableTouch: true,
    }),
    cookie: {
      maxAge: 1000 * 60 * 60 * 24, // 10 years
      httpOnly: true,
      sameSite: "lax", // csrf
      secure: false, // cookie only works in https
      domain: '.hotts.org'
    },
    saveUninitialized: false,
    secret: process.env.SESSION_SECRET,
    resave: false,
  })
);
```

Figure 16. Enable redis server

Subsequently, a GraphQL endpoint is then added to the file with ApolloServer.As shown in figure 17, the server must include a schema which describes all of the resolver to run functions that the client can connect to get and manipulate data from data which will be further developed in the later section. The Redis session is defined so that data can also be stored when the graphQL server is used. The GraphQL server endpoint will be "/graphql", for example, if the server runs on the port 4000, then the server will be on "http://localhost:4000/" and the graphql server will be on "http://localhost:4000/graphql" shown in figure 18 below.

```
const apolloServer = new ApolloServer({
  schema: await buildSchema({
    resolvers: [CommentResolver, UserResolver, PostResolver],
    validate: false,
  }),
  context: ({ req, res }) => ({
    req,
    res,
    redis,
    userLoader: createUserLoader(),
    updootLoader: createUpdootLoader(),
  }),
});
```

Figure 17. Apollo server initialize function

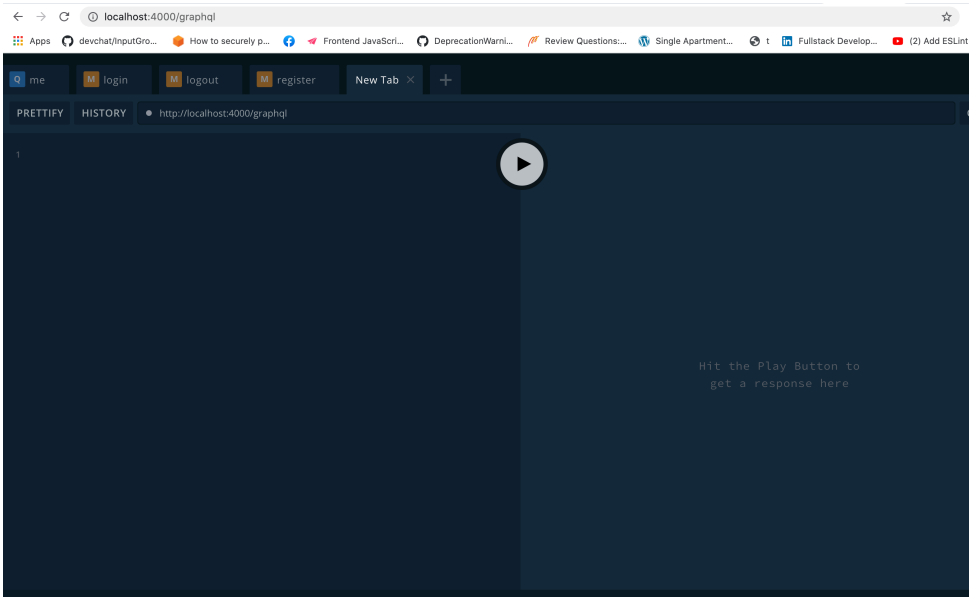

Figure 18. Graphql server

4.1.2 Writing Entity

An entity in typeORM is just a model instance that represents a table and the attribute for the field in the table that is on the database. There are 4 entities for this application: User, Post, Updoot, Comment.

4.1.2.1 User Entity

The figure 19 below displays the user entity indicating what data a developer can expect when the user interacts with the web application. The value of the column can be defined as string, date, Boolean, number or another entity. The field can be defined as mandatory data or optional by adding "!" next to the field name. The field can be defined as unique as the username and email field below. Every entity has an "ID" column used as a primary column, a mandatory column that cannot deleted and cannot have the same value. It is usually used as a unique identifier. There are some column that classify as "one to many", a single user can create multiple posts, create multiple comment and the user able to rate multiple comments and posts. The rating entity is called "Updoot"

here. There is some special type for “createdAt” and “updatedAt” field in typeORM and it will automatically keep track when the data is changed. TypeORM has an “ObjectType” function which turns the entity class into a graphql type object, then it can be later used for resolver.

```
@ObjectType()
@Entity()
export class User extends BaseEntity {
  @Field()
  @PrimaryGeneratedColumn()
  id!: number;

  @OneToMany(() => Post, (post) => post.creator)
  posts: Post[];

  @OneToMany(() => Updoot, (updoot) => updoot.post)
  updoots: Updoot[];

  @OneToMany(() => Comment, (comment) => comment.user)
  comments: Comment[];

  @Field()
  @Column({ unique: true })
  username!: string;

  @Field()
  @Column({ unique: true })
  email!: string;

  @Column()
  password!: string;

  @Field(() => String)
  @CreateDateColumn()
  createdAt: Date;

  @Field(() => String)
  @UpdateDateColumn()
  updatedAt: Date;
}
```

Figure 19. User entity

4.1.2.2 Post Entity

As the Post entity exhibited in figure 20, a single post can be rated by many users and can have multiple comment on it, but multiple post can be created by a single user. Then it is classified as a “many to one”. It also has a “createId” field to indicate the user.

```
@ObjectType()
@Entity()
export class Post extends BaseEntity {
  @Field()
  @PrimaryGeneratedColumn()
  id!: number;

  @Field()
  @Column()
  creatorId: number;

  @Field()
  @ManyToOne(() => User, (user) => user.posts)
  creator: User;

  @OneToMany(() => Updoot, (updoot) => updoot.user)
  updoots: Updoot[];

  @OneToMany(() => Comment, (comment) => comment.post)
  comments: Comment[];

  @Field(() => Int, { nullable: true })
  voteStatus: number | null; //1 or -1 or null

  @Field()
  @Column()
  title!: string;

  @Field()
  @Column()
  text!: string;

  @Field()
  @Column({ type: "int", default: 0 })
  points!: number;

  @Field(() => String)
  @CreateDateColumn()
  createdAt: Date;

  @Field(() => String)
  @UpdateDateColumn()
  updatedAt: Date;
}
```

Figure 20. Post entity

4.1.2.3 Updoot Entity

As shown on figure 21 below, the Updoot entity can have many users and many users can rate a post. In case if a user wanted to delete a Post, the “Updoot” entity for the “Post” entities has to be deleted first or it will violate a foreign key constraint. typeORM has a function called onDelete: “CASADE”, upon deleting a post,the “Updoot” entities will automatically be deleted.

```
@ObjectType()
@Entity()
export class Updoot extends BaseEntity {
  @Column({ type: "int" })
  value: number;

  @PrimaryColumn()
  userId: number;

  @ManyToOne(() => User, (user) => user.updoots)
  user: User;

  @PrimaryColumn()
  postId: number;

  @ManyToOne(() => Post, (post) => post.updoots, {
    onDelete: "CASCADE",
  })
  post: Post;
}
```

Figure 21. Updoot entity

4.1.2.4 Comment Entity

As show on figure 22 below, the comment entity is very simple. It has textfield, “postId” to indicate the post, “userId” to indicate the user. Many comments can be created by a user and many comments can be created for a post.

```
@ObjectType()
@Entity()
export class Comment extends BaseEntity {
  @Field()
  @PrimaryGeneratedColumn()
  id!: number;

  @Field()
  @Column({ type: "text" })
  text: string;

  @Field()
  @PrimaryColumn()
  userId: number;

  @Field()
  @ManyToOne(() => User, (user) => user.comments)
  user: User;

  @Field()
  @PrimaryColumn()
  postId: number;

  @Field()
  @ManyToOne(() => Post, (post) => post.comments )
  post: Post;

  @Field(() => String)
  @CreateDateColumn()
  createdAt: Date;

  @Field(() => String)
  @UpdateDateColumn()
  updatedAt: Date;
}
```

Figure 22. Comment entity

4.1.3 Writing Resolver

Resolvers functions resolve the field and type in a schema. It takes in an argument and an instruct query or a mutation to return the exact data types in the schema. A Graphql query used to fetch and read a value from the database while a mutation is used for writing or posting data into a database. As display in figure 23, the query is required to return a value with a “User” object type or null. Inside the query, the “User” entities run the data mapper pattern “findOne” to find a specific value that has the same as the “ID” argument input value in the table “User” in the postgres database and return a value according to the “User” object type.

```
@Query(() => User, { nullable: true })
async userById(@Arg("id", () => Int) id: number, @Ctx() { req }: MyContext) {
  return await User.findOne(id);
}
```

Figure 23. Query function example

4.1.3.1 Authentication and authorization

The user resolver is handling the authentication for the project. For security, a user password has to go through encryption before being uploaded into the database. The “argon2” library is installed to handle the encryption. It was also selected as the winner of Password Hashing Competition in 2015 which adds to it credibility.

```
const hashedPassword = await argon2.hash(options.password);
```

Figure 24. Hash password using argon2

As shown on figure 24, when a user logs in, the password given will be compared using a function “verify” with the encrypted password gotten from the database with argon2 to determine if the password is correct.

```
const valid = await argon2.verify(user.password, password);
```

Figure 24. Verify password using argon2

After the password is verified, the user ID gotten from the database is stored into the Redis session for authentication shown in figure 25. As shown in figure 26, create a middleware function need to

be created to check for authentication when performing a query or a mutation. If there are no "userId" value in the sessions then an error is thrown instead.

```
req.session.userId = user.id;
```

Figure 25. Store data in Redis session

```
export const isAuth: MiddlewareFn<MyContext> = ({ context }, next) => {
  if (!context.req.session.userId) {
    throw new Error("not authenticated");
  }
  return next();
};
```

Figure 26. Middleware function check for authentication

As depicted in figure 27 below, an authenticated function, such as create, edit, delete post, will have the authentication middle function applied.

```
@Mutation(() => Post)
@UseMiddleware(isAuth)
async createPost(
  @Arg("input") input: PostInput,
  @Ctx() { req }: MyContext
): Promise<Post> {
  return Post.create({
    ...input,
    creatorId: req.session.userId,
  }).save();
}
```

Figure 27. Apply middleware authentication function

4.2 Front end implementation

The development of the frontend include initialization, integration of Urql GraphQL client, handling cache with Urql Graphcache, setting up server side rendering and the development of different pages in the forum.

4.2.1 Front end set up

The web application is built in Next.js (React). The command in figure 28 is used to create a new default Next.js template, with everything set up ready.

```
yarn create next-app
```

Figure 28. Command to create Next.js application

For the application UI implementation, the application uses ChakraUI, a simple component library that has a pre-built React component that is easy to use, reusable and composable and runs the command from figure 29 to be installed.

```
yarn add @chakra-ui/react framer-motion
```

Figure 29. Command to add chakra-ui library

For the client frontend to make a GraphQL request, it is needed to set up an urql GraphQL client by using the command from figure 30.

```
yarn add urql graphql
```

Figure 30. Command to add urql graphql

To set up server side render just run the command from figure 31 and then wrap any page with withUrqlClient higer-componet, to enable server side render for the page just write { ssr: true } after the urql client.

```
export default withUrqlClient(createUrqlClient, { ssr: true })(CreatePost);
```

Figure 31. Enable urql client and server-side rendering

After that, an Urql client function “createUrqlClient” is called to handle the graphql request, Redis cache and cookie. As shown from figure 32, the GraphQL request from “http://localhost:4000/graphql”, cacheExchange from urql allows the developer to have custom update cache whenever a mutation or a query is called.

```
export const createUrqlClient = (ssrExchange: any, ctx: any) => {
  let cookie = "";
  if (isServer()) {
    cookie = ctx?.req?.headers?.cookie;
  }
  return {
    url: "http://localhost:4000/graphql",
    fetchOptions: {
      credentials: "include" as const,
      headers: cookie
        ? {
            cookie,
          }
        : undefined,
    }, //send cookie -> need cookies to register
    exchanges: [
      dedupExchange,
      cacheExchange({
        keys: {
          PaginatedPosts: () => null,
          PaginatedComments: () => null,
        },
        resolvers: {
          Query: {
            posts: cursorPagination(),
            getCommentByPostId: commentPagination(),
          },
        },
        updates: {
          Mutation: {
            deletePost: (_result, args, cache, info) => {
              cache.invalidate({
                __typename: "Post",
                id: (args as DeletePostMutationVariables).id,
              });
            },

            vote: (_result, args, cache, info) => {
              const { postId, value } = args as VoteMutationVariables;
              const data = cache.readFragment(
                gql`
                  fragment _ on Post {
                    id
                    points
```

Figure 32. Function to create Urql client

4.2.2 Page routing

Routing is automatically handled with Next.js. There is a folder called pages, If a folder or a file is created in the pages folder, then it will automatically create a route with the file name shown in figure 33, 34 and 35.

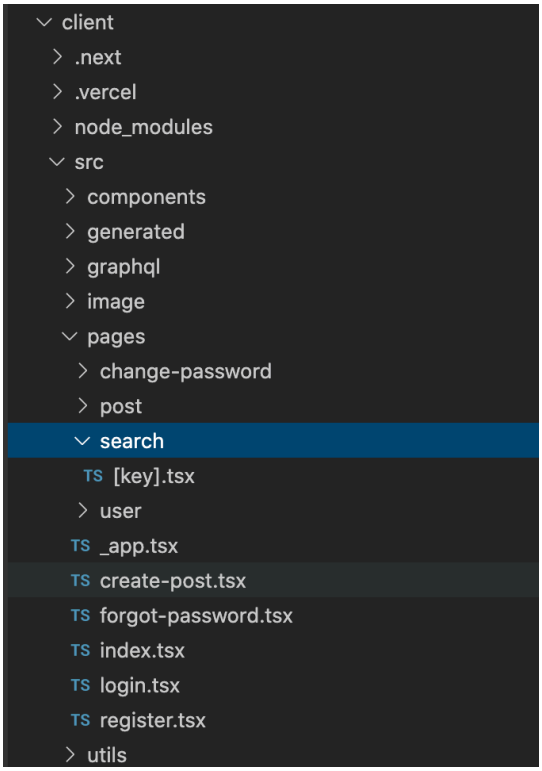

Figure 33. Next.js page router folder

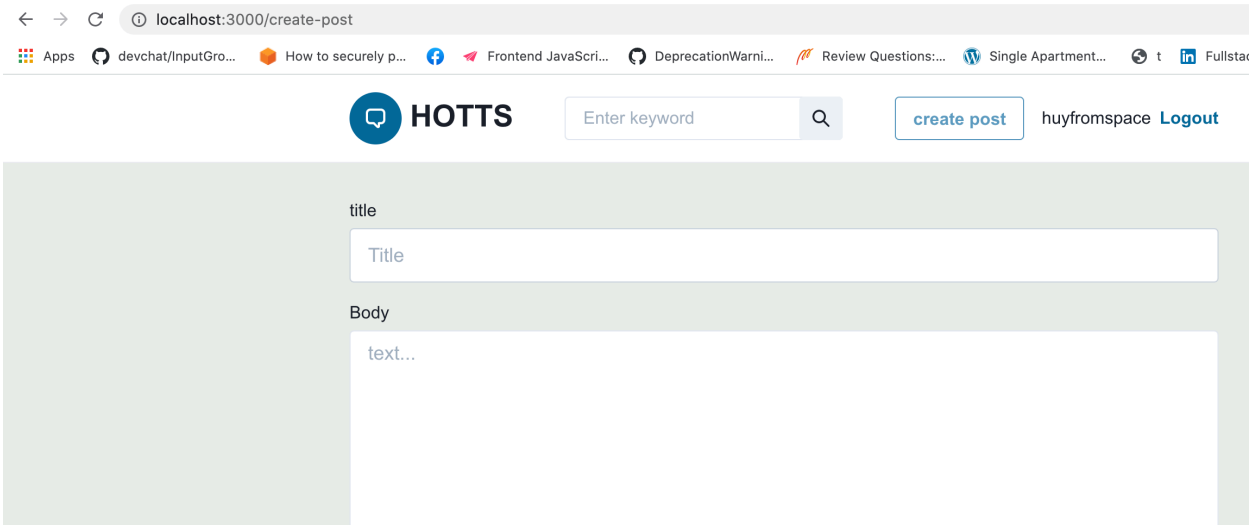

Figure 34. Next.js page router in browser

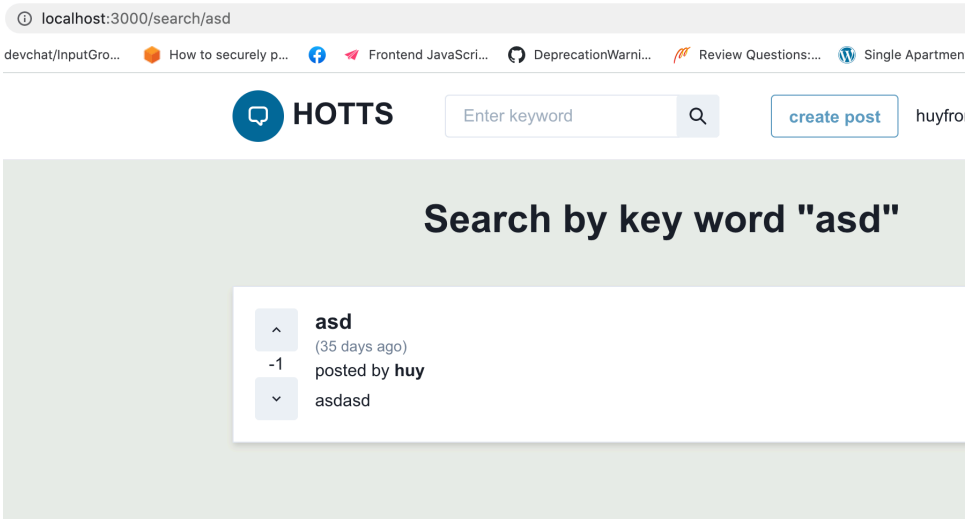

Figure 35. Multiple folder next.js page router in browser

4.2.3 GraphQL code generator

There is a GraphQL code generator library “graphql-codegen” which allows the developer to generate code for the frontend client from the GraphQL schema and operation using a single function. The library can be added using the command in figure 36. This can save the developer lots of time by saving them time to manually implement the query and mutation as well as redefine all the types of the data return.

```
yarn add -D @graphql-codegen/cli
```

Figure 36. Command to add graphql code generator library

Then to set up everything using GraphQL code generator by running the command in figure 37, it provide question to ask which environment the application is built on and after everything it will generate a codegen.yml file

```
yarn graphql-codegen init
```

Figure 37. Command to set up graphql code generator

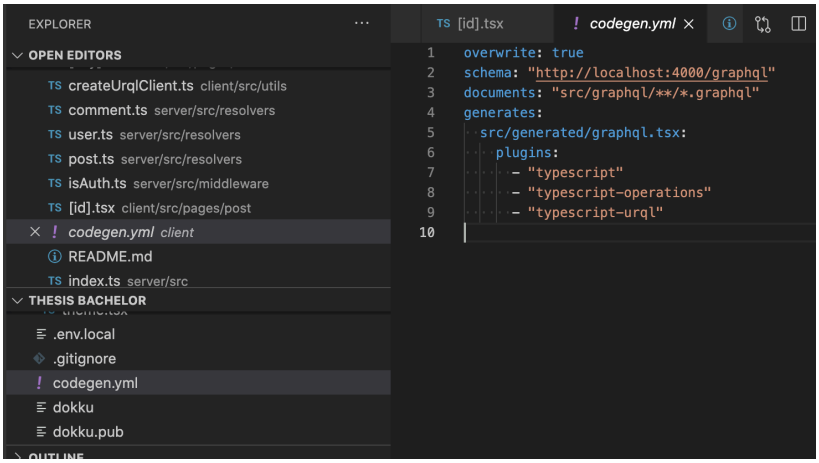

Figure 38. Codegen.yml file

As depicted in figure 38 above, it takes the GraphQL schema in "http://localhost:4000/graphql" and it will take any ".graphql" file in the folder "src/graphql" and automatically generate a mutation, query function and type.

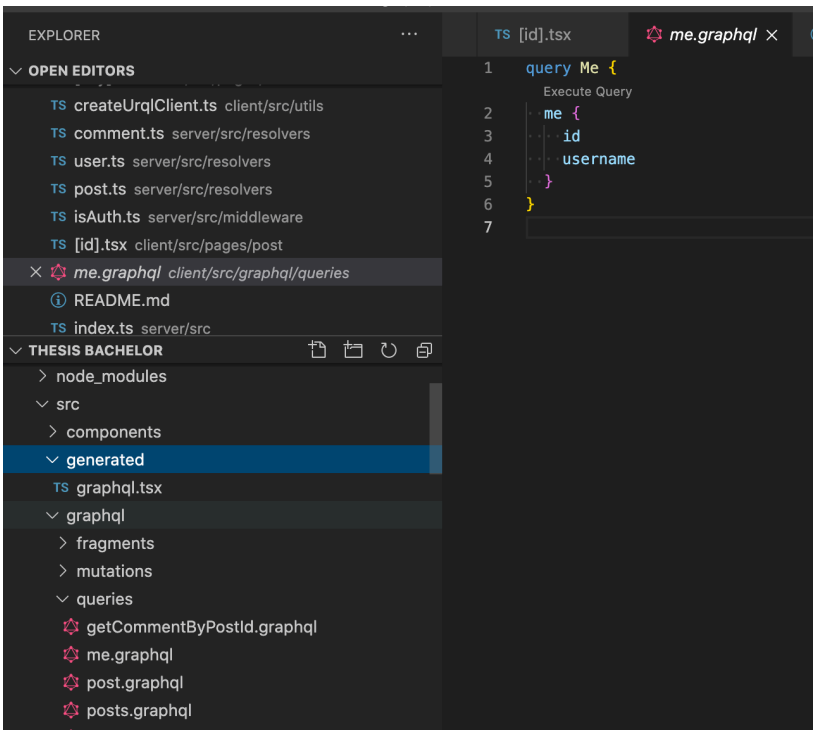

Figure 39. Graphql code generator example file

As depicted in figure 39, a query and mutation in a ".graphql" file are written quite similar to how they are written on the graphql server, Then the command "yarn graphql-codegen --config co-degen.yml" is run and it will be executed into the code generator "codegen.yml" file and generated

into the file “generated.graphql.tsx” for all the query and mutation is ready for client frontend to used.

4.2.4 Login, register pages and authorization process

The Register page and login pages shown in figure 40 and 41 are pages that create and log into an account for the user to do tasks that need authentication such as create, edit, delete posts, make comment, rate posts. Both pages have the same coding structure. The only difference is the GraphQL mutation.

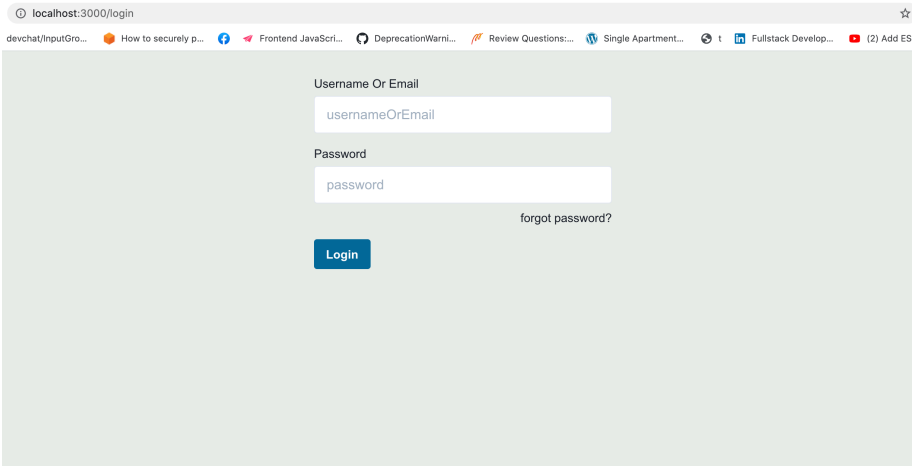

Figure 40. Login pages

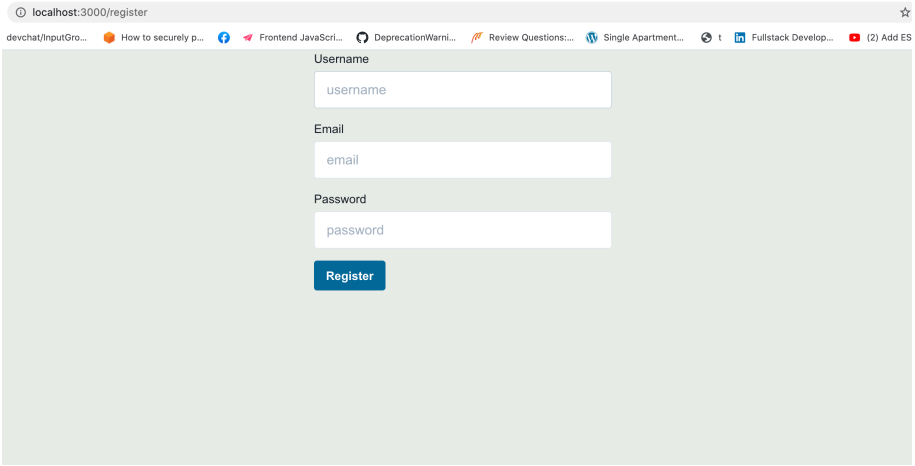

Figure 41. Register pages

For registers and login pages, they use the chakraUI “Formik” form to handle and get all the input values from the input components to use in the “onSubmit” function in figure 42 when the submit

button is clicked. A Graphql mutation can be imported from a code generate file “generated/graphql.tsx” to be used and Next.js also provides with a router imported from “next/router” for navigation on the application as shown in figure 43.

```
<Formik
  initialValues={{ usernameOrEmail: "", password: "" }}
  onSubmit={async (values, { setErrors }) => {
    const response = await login(values);
    if (response.data?.login.errors) {
      setErrors(toErrorMap(response.data?.login.errors));
    } else if (response.data?.login.user) {
      router.push(
        typeof router.query.next === "string" ? router.query.next : "/"
      );
    }
  }}
>
  {({ isSubmitting }) => (
    <Form>
      <InputField
        name="usernameOrEmail"
        label="Username Or Email"
        placeholder="usernameOrEmail"
      />
      <Box mt={4}>
        <InputField
          name="password"
          label="Password"
          placeholder="password"
          type="password"
        />
      </Box>
      <Flex mt={2}>
        <NextLink href="/forgot-password">
          <Link ml="auto">forgot password?</Link>
        </NextLink>
      </Flex>

      <Button
        type="submit"
        backgroundColor="#02699c"
        color="white"
        mt={4}
        isLoading={isSubmitting}
      >
        Login
      </Button>
    </Form>
  )}
</Formik>
```

Figure 42. Login form

```
import { Box, Button, Flex, Link } from "@chakra-ui/core";
import { Form, Formik } from "formik";
import { withUrqlClient } from "next-urql";
import NextLink from "next/link";
import { useRouter } from "next/router";
import React from "react";
import { InputField } from "../components/InputField";
import { Wrapper } from "../components/Wrapper";
import { useLoginMutation } from "../generated/graphql";
import { createUrqlClient } from "../utils/createUrqlClient";
import { toErrorMap } from "../utils/toErrorMap";

export const Login: React.FC<{}> = ({}) => {
  const router = useRouter();
  const [, login] = useLoginMutation();
```

Figure 43. import router and login mutation

To help reduce the bandwidth or network traffic and less queries need to be made into the server, it can be solved storing data in cache and when the same query is made, the data is already in cache and can be reused. To update the logged in user information, cache can be updated with a

“cacheExchange()” function inside a “createUrqlClient” function. As shown in figure 44 below, after the GraphQL is made, this function will check for a valid login, then the cache ispopulated with user information upon success.

```
login: (_result, args, cache, info) => {
  betterUpdateQuery<LoginMutation, MeQuery>(
    cache,
    { query: MeDocument },
    _result,
    (result, query) => {
      if (result.login.errors) {
        return query;
      } else {
        return {
          me: result.login.user,
        };
      }
    }
  );
},
```

Figure 44. Cache updated logic

When a user uses an unauthorized function, they need to be redirected to the sign in to show them that they need to log in to use the feature. As shown in figure 45, it is a function to check for authentication for a user only interaction or page access. It will run a query that checks user info from the Redis service and if the query does not return a user, then it will redirect the user to the login page.

```
export const useIsAuth = () => {
  const [{ data, fetching }] = useMeQuery();
  const router = useRouter();

  useEffect(() => {
    if (!fetching && !data?.me) {
      router.replace("/login?next=" + router.pathname);
    }
  }, [fetching, data, router]);
};
```

Figure 45. Check authentication function

4.2.5 Home page

As shown in figure 46, the Home page by default displays all posts sorted by the post upload data. The posts component gets all the posts from the GraphQL posts query and all the posts will be mapped and displayed as a multiple small post component. The information displayed in each post component will check for authorization. If the post is created by the same user in the cache, then it will allow the user to edit or delete the post. If the user has already rated the post, then it will display the options the user had chosen.

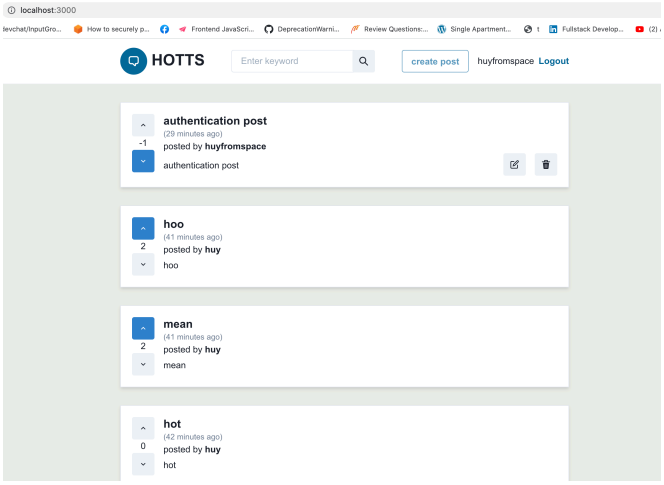

Figure 46. Home page

The problem in displaying all the posts at once is that it will cost a long load time for the client if there are lots of posts on the server. A pagination function is needed for the posts component for the website to run smoothly with a low down time. As shown in figure 47, the posts query takes in 4 arguments, the cursor which will make the query fetch post that is created after a certain date, the limit which limit the maximum amount of posts that can be fetched in a single query, the keyword which will fetch posts that have a name or an author associated to the keyword, the creator ID which fetches posts according to the user ID.

```
query Posts(
  Execute Query
  $limit: Int!
  $cursor: String
  $keyword: String
  $creatorId: Int
) {
  posts(
    cursor: $cursor
    limit: $limit
    keyword: $keyword
    creatorId: $creatorId
  ) {
    hasMore
    id
    createdAt
    updatedAt
    title
    text
    points
    voteStatus
    textSnippet
    creator {
      id
      username
    }
  }
}
```

Figure 47. Posts query

A Posts limit is set for each query and if there are still more posts in the database, then it will return a value “hasMore:true” and the posts component will display an option to load more posts as shown in figure 48.As depicted in figure 49, How the pagination works is that the user clicks the “Load More” button, then it will make another query but it sets the cursor to the last posts upload date, then the query will fetch posts that are created after the last posts.

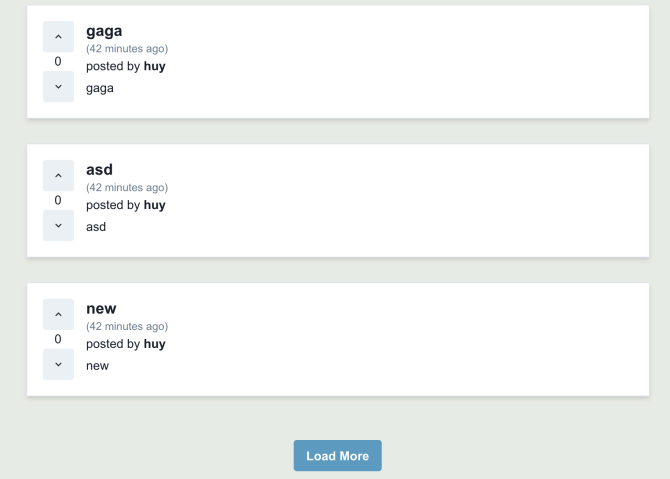

Figure 48. Load more option

```
<Button
  onClick={() => {
    setVariables({
      limit: variables.limit,
      cursor: data.posts.posts[data.posts.posts.length - 1].createdAt,
      keyword,
      creatorId: null,
    });
  }}
  isLoading={fetching}
  m="auto"
  my={4}
  backgroundColor="#5c9dc0"
  color="white"
>
  Load More
</Button>
```

Figure 49. Posts pagination logic

After that the new posts fetched from the query need to be updated in the cache along with the posts from last posts query to create a continuation lists of posts. As shown in figure 50, the function “cursorPagination” will check for the query and the argument passes into them to check if the query fetches all posts, posts by user or posts by keyword and it will look into the cache the existing data for the query and it will populate the lists with data from the previous query with new data from the new query. The posts pagination function is complete, when the user wants more posts to load, the posts data will not disappear with page reload or component re-render. This function also works for search.

```
export const cursorPagination = (): Resolver => {
  return (_parent, fieldArgs, cache, info) => {
    const { parentKey: entityKey, fieldName } = info;
    const allFields = cache.inspectFields(entityKey);
    const fieldInfos = allFields.filter((info) => info.fieldName === fieldName);
    const size = fieldInfos.length;
    if (size === 0) {
      return undefined;
    }
    const fieldKey = `${fieldName}(${stringifyVariables(fieldArgs)})`;
    const isItInTheCache = cache.resolve(
      cache.resolveFieldByKey(entityKey, fieldKey) as string,
      "posts"
    );
    info.partial = !isItInTheCache;
    let hasMore = true;

    const results: string[] = [];
    fieldInfos.forEach((fi) => {
      if (fieldArgs.keyword && fi.arguments?.keyword === fieldArgs.keyword) {
        //cache for different keyword search queery
        const key = cache.resolveFieldByKey(entityKey, fi.fieldKey) as string;
        const data = cache.resolve(key, "posts") as string[];
```

```
      const _hasMore = cache.resolve(key, "hasMore");

      if (!_hasMore) {
        hasMore = _hasMore as boolean;
      }
      results.push(...data);
    } else {
      if (
        (fieldArgs.creatorId &&
          fi.arguments?.creatorId === fieldArgs.creatorId &&
          !fieldArgs.keyword &&
          !fi.arguments?.keyword) ||
        (!fieldArgs.creatorId &&
          !fi.arguments?.creatorId &&
          !fieldArgs.keyword &&
          !fi.arguments?.keyword)
      ) {
        //cache for different keyword search queery
        const key = cache.resolveFieldByKey(entityKey, fi.fieldKey) as string;
        const data = cache.resolve(key, "posts") as string[];
        const _hasMore = cache.resolve(key, "hasMore");
        if (!_hasMore) {
          hasMore = _hasMore as boolean;
        }
        results.push(...data);
      }
    }
  });
  return {
    __typename: "PaginatedPosts",
    hasMore,
    posts: results,
  };
};
};
```

Figure 50. Posts pagination cache logic

4.2.6 Navigation bar

A navigation bar is the bar at the top of the application that displays user information and navigates the user to pages to perform tasks. The navigation bar will check the cache for user information. If there is no user data, then it will show the user an option to go to the Login and Register page as illustrated in figure 51 and if there is user data then it will display the user name and an option to logout, which removes the user data from cache as show in figure 52. There is a search bar on the

navigation bar which allows the user to search for the post name or the author of the post. It just makes a new query and returns a new list of data with the keyword provided.

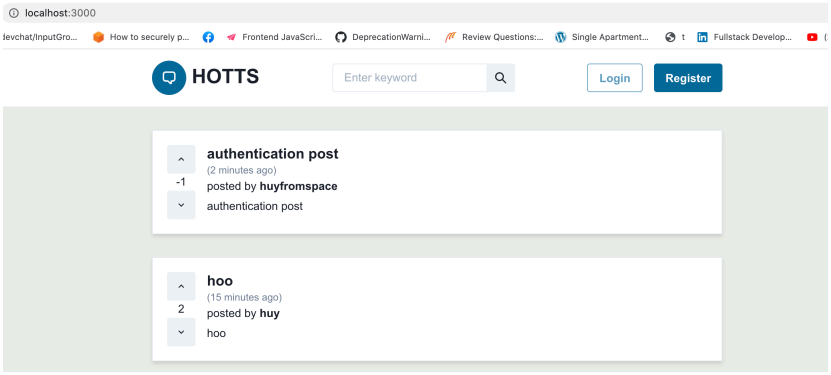

Figure 51. Default navigation bar

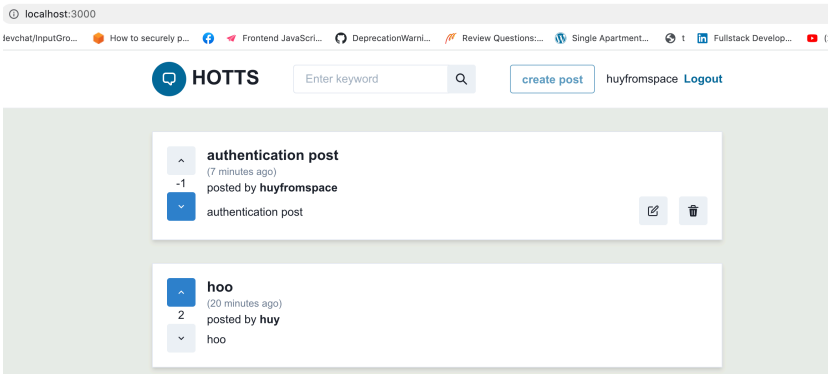

Figure 52. Navigation bar with logged in user

4.2.7 Create, edit, delete post

A Create and Edit post includes a form shown in figure 53 below. It consists of a title and a body field. When a post is created or edited it will call a mutation to add or change post data in the database. The edit Post page also queries the post current information and populates the field. The delete post button just calls a mutation to delete the data in the database.

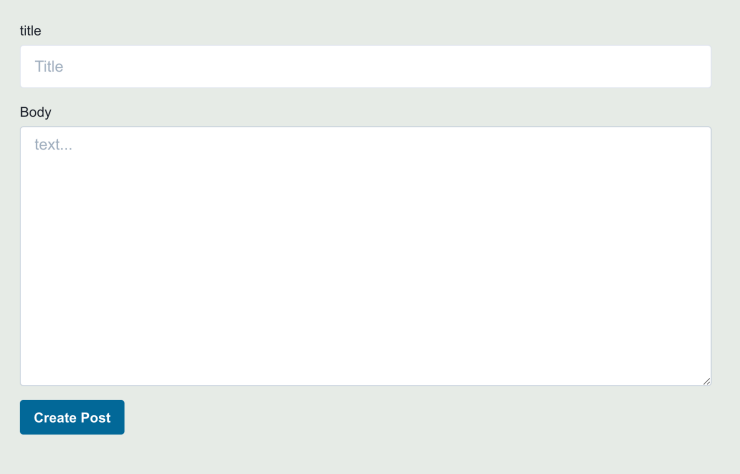

Figure 53. Create, edit form

Since the posts are stored in cache the post data will not disappear. In figure 54 is a function which is called every time anything changes regarding all the posts data to clear the cache data so that the application can query new data and update the web application. The function will check for all the fields in the cache and find the “posts” query data and it will go through everything in the field and clear the data.

```
function invalidateAllPosts(cache: Cache) {
  const allFields = cache.inspectFields("Query");
  const fieldInfos = allFields.filter((info) => info.fieldName === "posts");

  fieldInfos.forEach((fi) => {
    cache.invalidate("Query", "posts", fi.arguments || {});
  });
}
```

Figure 54. Function to clear post cache data

4.2.8 Single post page and comment

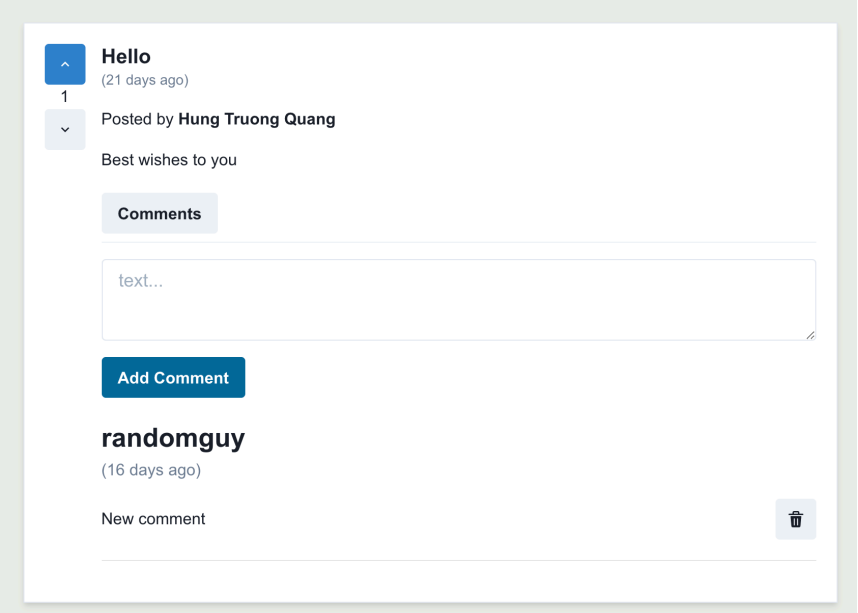

Figure 55. Single post page

The single post page shows the post data and the post full text. It also has the posts component from the homepage to show other posts below. All the rating function, edit and delete still remain the same on the homepage but it provides a comment section for the user to interact. After populating the comment field with text and clicking “Add comment”, it will make a mutation to the server to create a comment. Same for the posts, the comments also clear the cache data upon a comment and it will re-fetch a comment from the query and display them like in figure 55 above and the comment can also be deleted by the comment’s creator

5 DEPLOYMENT

5.1 Backend deployment

The back end is deployed on Digital Ocean, a cloud computing platform that provides cloud services to developers to help them deploy and scale their application. It is one of the most popular cloud services alongside with Amazon Web Service and Microsoft's Azure. The application run on Digital Ocean will run on a Virtual Machine called "droplet". The developer can choose the size of the "droplet" and choose which region the data will run on.

In order to set up the server, it can be deployed via Dokku, an open-source Platform, as a Service and it is easy to run and it can be set up through the developer's private virtual service (Digital Ocean). First the user creates an account and logs into Digital Ocean. Then they select marketplace Dokku, going into the cloud page shown in figure 56 and choose the basic plan with $5 per months as the cheapest option and request an SSH key, then finally create an image.

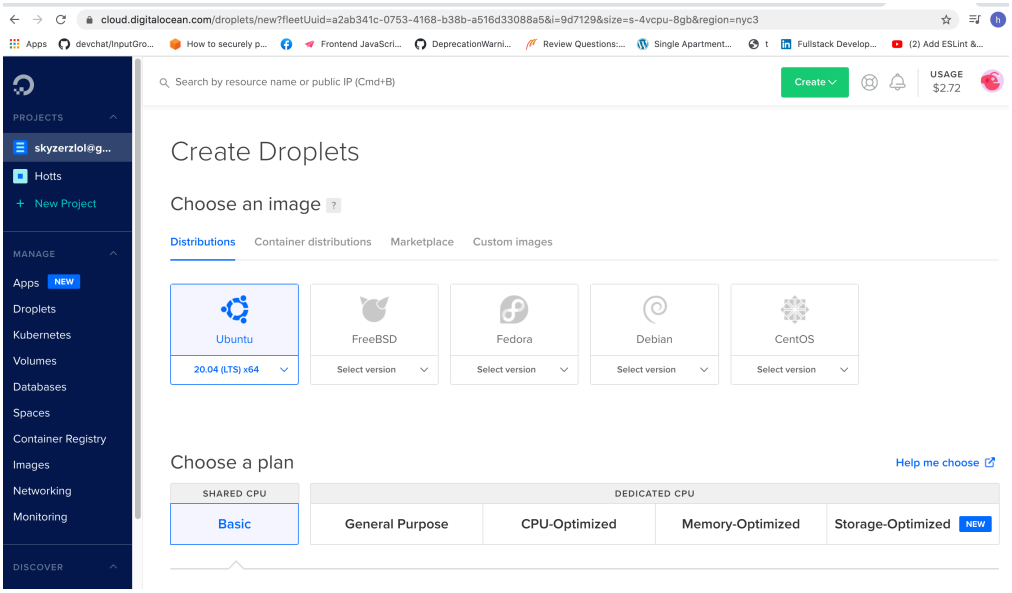

Figure 56. Digital Ocean cloud page

After that, go to the application and install Dokku with the command from figure 57.

```
sudo dokku plugin:install https://github.com/dokku/dokku-postgres.git
```

Figure 57. Command to install dokku

Then the user creates a postgres database on Dokku with the command in figure 58, creates and link a Redis cache to the database just like in figure 59.

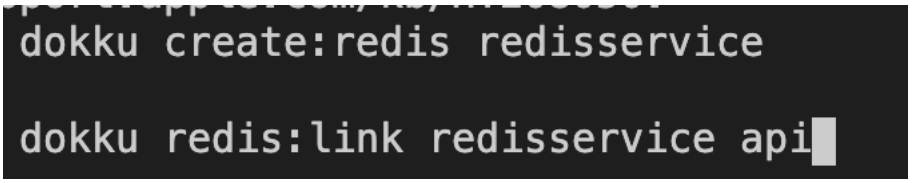

Figure 58. Command to create postgres database

Whenever the server is updated or deployed, it has a Dockerfile that will instruct to command in the server to install dependencies, build the application and ask them to use a production environment variable for sensitive information, such as a secrets session key, database URL conain name and password. The dockerfile content for the application is in figure 59.

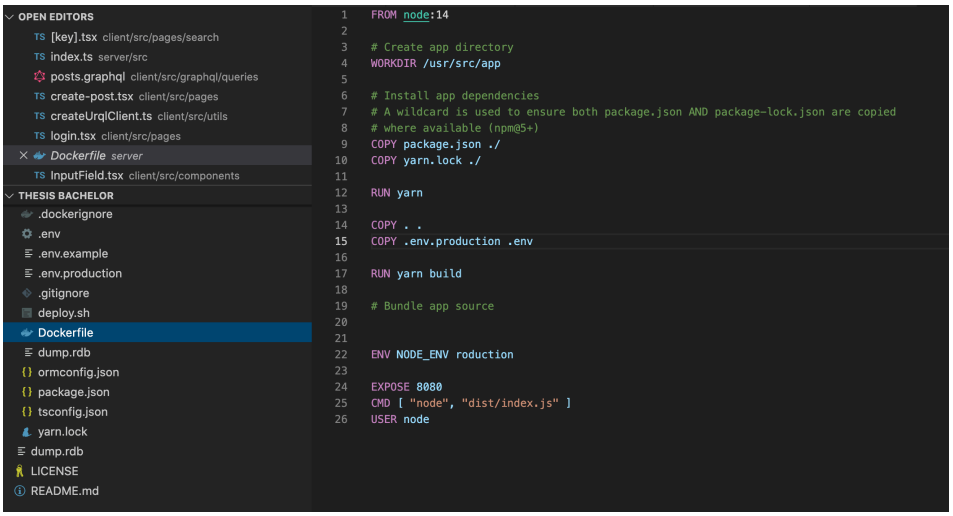

Figure 59. Dockerfile

Finally, the users create a Doker account on the Docker hub and logs into the account on the terminal. To run the Digital Ocean virtual server referred as a “droplet”, the user simply runs the SSH provided on the Digital Ocean cloud site for the server. The developer can then build an image using docker shown in figure 60. Then they can deploy the Docker image to Digital Ocean according to the instruction from figure 61. Then provide the Digital Ocean “droplet” with a domain, then the domain URL will provide an API for the frontend to use.

figure 60. Docker build image command

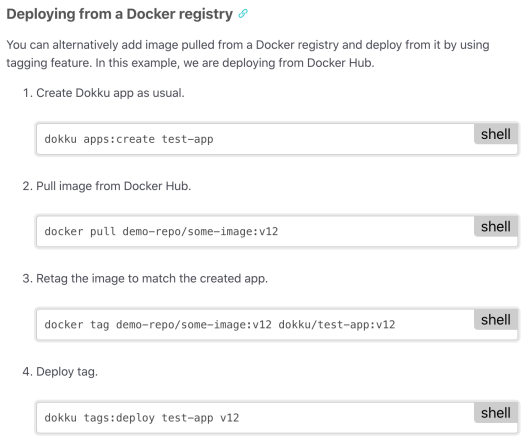

figure 61. Docker Image deployment

A domain purchase for this application is from namecheap.com. Then the “dokku-letsencrypt” library is used to automatically retrieve and install a TLS certificate, which helps encrypts data and send it over the Internet to prevent hackers to get the data. The command is run from figure 62 to check for domains app vhost, domains global vhost and configure the domains accordingly to the domain purchased.

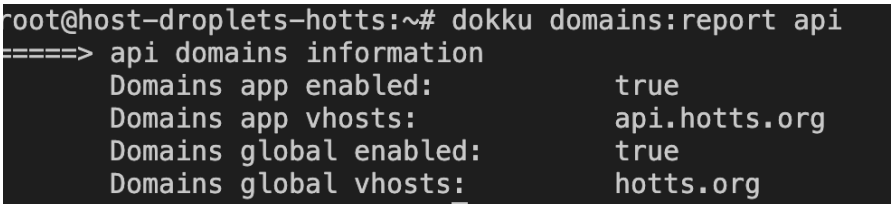

Figure 62. Command for Dokku api domains

5.2 Frontend deployment

To deploy the frontend, the application will be using Vercel, the creator of Next.js allows the developer to host static sites with serverless functions. This ables the user to quickly deploy and scale automatically with only a few configurations. Vercel will split up the Virtual Private Server with frontend that ables to take in a request and the API backend to return the data. Vercel libraries should be installed for the frontend. An account is needed to use Vercel. It can be created on the website and it is logged in with the terminal. Then a simple run command “vercel” is run for the frontend applications so it can push the latest application build to productions. On the Vercel website, the user just provides the application with an API URL for environment variables for the frontend. Finally, domain is provided with an IP address for frontend and backend, then the URL redirect link for the website application is shown in figure 63 and then the website is complete.

Type	Host	Value
A Record	@	76.76.21.21
A Record	api	139.59.133.213
URL Redirect Record	www	https://hotts.org

Figure 63. Domains configuration in Namecheap

CONCLUSION

The thesis aimed to analyze and describe modern JavaScript technology in web development and how to build and deploy a website based on it. A great amount of time was put into research on the topic of full stack JavaScript application and learning different framework such as React, Node.js, Express, GraphQL and Postgres. The advantages and disadvantages of all technology involved in the application were examined and compared to other similar technologies. Moreover, the speed and performance of the application were also acknowledged. These technologies were applied to build and deploy a discussion forum where people can share their opinions. The end product is deployed for public used and serve as a prototype to show how these technologies have practical used.

Conclusion, JavaScript has become the most popular programming language in web development for its flexibility to implemented in both backend and frontend while provides with high performance and speed. With a huge community to support it, full stack JavaScript application has become easier to learn and implemented than ever before. However, full stack JavaScript applications is not option to use for every web application since Node.js are unable to handle data heavy related application. There are far better technologies that can be used to handle projects like machine learning, algorithms, or heavy mathematical calculations. Even so, full stack JavaScript application is still considered a young technology and with a huge community of developer to supports, JavaScript technologies can only evolve with time.

REFERENCES

1. HOW THE JSX TRANSFORM WORKS? [Internet]. jaketrent.com. Available from: https://jaketrent.com/post/how-jsx-transform-works/
2. Introduction to the DOM? [Internet]. mozilla.org. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction
3. DOM (Document Object Model). [Internet]. geeksforgeeks.org. Available from: https://www.geeksforgeeks.org/dom-document-object-model/
4. React Functional Components VS Class Components. [Internet]. medium.com. Available from: https://medium.com/wesionary-team/react-functional-components-vs-class-components-86a2d2821a22
5. Examples- Quick Compare? [Internet]. tkssharma.gitbook.io. Available from: https://tkssharma.gitbook.io/react-training/day-03/type-of-react-components/examples-quick-compare
6. Understanding React Hooks? [Internet]. serverless-stack.com. Available from: https://serverless-stack.com/chapters/understanding-react-hooks.html
7. Client Side Rendering Vs Server Side Rendering in React js & Next js. [Internet]. medium.com. Available from: https://yudhajitadhikary.medium.com/client-side-rendering-vs-server-side-rendering-in-react-js-next-js-b74b909c7c51
8. Why The Hell Would I Use Node.js? A Case-by-Case Tutorial. [Internet]. toptal.com. Available from: https://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js#:~:text=js%20used%20for%3F-,Node.,push%2Dbased%20architectures%20in%20mind
9. How Node JS middleware Works?. [Internet]. medium.com. Available from: https://medium.com/@selvaganesh93/how-node-js-middleware-works-d8e02a936113
10. What is PostgreSQL? How Does PostgreSQL Work? [Internet]. tecmint.com. Available from: https://www.tecmint.com/what-is-postgresql-how-does-postgresql-work/#:~:text=PostgreSQL%20uses%20a%20client%2Dserver,the%20database%20from%20client%20applications
11. Distributed caching in Redis. [Internet]. medium.com. Available from: https://medium.com/@123williams93/distributed-caching-in-redis-8ff882bf79ac

12. A Beginner's Guide: Setting Up Your Development Environment. [Internet]. medium.com. Available from: https://medium.com/@mbyfieldcameron/a-beginners-guide-to-setting-up-your-development-environment-7d3a38b3656d
13. How it works. [Internet]. atlassian.com. Available from: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
14. Three Reasons to Use Yarn in 2020 (and Beyond). atomicobject.com. Available from: https://spin.atomicobject.com/2020/03/15/why-yarn-2020/