УНИВЕРЗИТЕТ „СВ. КИРИЛ И МЕТОДИЈ“ – СКОПЈЕ

ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И
КОМПЈУТЕРСКО ИНЖЕНЕРСТВО – СКОПЈЕ

НАСОКА: КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

СЕМИНАРСКА РАБОТА

ПРЕДМЕТ:

Напреден веб дизајн

ТЕМА:

Форум

ментор:

д-р Бобан Јоксимоски

кандидат:

Мартин Трајковски 195063

Февруари 2022

Содржина

1. Вовед

Оваа апликација се состои од два делови Frontend и Backend:

Frontend делот се фокусира за визуелниот приказ и интеракцијата на различни елемент со корисникот може и да се вика Client side оваа апликација за Frontend ќе користи Vue што е Frontend framework и Tailwind со кој што ке биде направено стилизирањето (CSS-от)

Backend делот се грижи за серверскиот дел на апликацијата преку кој се земаат податоците или се додаваат податоци во некоја база исто така се нарекува Server side оваа апликација за Backend делот ќе користи Mongoose за базата и GraphQL преку кој ке се земаат податоците од Mongoose базата и Node.js со кој се стартува серверот.

2. Vue

Како што е кажано за Vue се користи за frontend-от на апликацијата за да се креира една ваква апликација се користи vue/cli со користење на командата vue create [project name]. Во оваа апликација се користи Vue router, Composition API и Vue PWA (Progressive Web App).

2.1 Vue Router

Овозможува на Vue да рутира помеѓу компонентите и која компонента ке се прикаже на страната. На слика 1 е прикажана router компонентата се состои од низа од објекти кои се кои се рути до различни компоненти,објектот {path:‘/communities’, name: ‘communitiesListView’, component: () => import(‘@/views/CommunitiesView.vue’) } овозможува да може да се користи рутата http://localhost/communties, name атрибутот овозможува да се користи route-link компонентата со атрибутот ‘to’ како на пример: <route-link :to=”{ name: ’communities ListView’}”></route-link> исто така објектот во routes може да има и атрибут children кој е низа со објекти од рути кој омозможува друга содржина на готов изглед template со користење на компонентата <router-view />

```
import { createRouter, createWebHistory } from 'vue-router';

const routes = [
  { path: '/', name: 'homeView', component: () => import('@/views/Home.vue') },
  {
    path: '/community/:id',
    name: 'communityDetailsView',
    component: () => import('@/views/CommunityDetails.vue'),
  },
  {
    path: '/communities',
    name: 'communitiesListView',
    component: () => import('@/views/CommunitiesList.vue'),
  },
  {
    path: '/register',
    name: 'registerView',
    component: () => import('@/views/RegisterView.vue'),
  },
  {
    path: '/login',
    name: 'loginView',
    component: () => import('@/views/LoginView.vue'),
  },
  {
    path: '/submit',
    name: 'createPostView',
    component: () => import('@/views/CreatePost.vue'),
  },
  {
    path: '/user/options',
    name: 'userOptionsView',
    component: () => import('@/views/UserOptionsView.vue'),
    children: [
      {
        name: 'account-options',
        path: 'account',
        component: () => import('@/views/UserOptionsAccount.vue'),
      },
      {
        name: 'notifications-options',
        path: 'notifications',
        component: () => import('@/views/UserOptionsNotifications.vue'),
      },
      {
        name: 'profile-options',
        path: 'profile',
        component: () => import('@/views/UserOptionsProfile.vue'),
      },
    ],
  },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});
```

Слика 1: Vue router

2.2 Креирање на Vue апликација

На слика 2 е прикажа главната компонента преку која се генерира темплејт за сите понатамошни изгледи преку <router-view /> копмонентата во script тагот се наога vue објектот кој се зема преку ‘import App from ‘./App.vue’ во main.js преку кој потоа се користи createApp.use(router).mount(‘#app);’ кој креира една Vue аппликација го поврзува рутер објектот кој го направивме претходно и се поврзува со елемент во html кој има CSS selector #app.

Во Vue објектот бидејќи ова е главна компонента тука мора да кажеме дека ке ја користиме NavbarComponent и затоа ја ставаме во components атрибут и се користи Composition API кое што омозможува преку наредбата provide да може да се инјектира понатаму во било која компонента некоја променлива или објект со нерадбата inject, тоа се прави преку setup() функцијата.

```
<template>
  <navbar-component></navbar-component>
  <div
    class="bg-gray-100 dark:bg-neutral-800 py-2 flex justify-center h-screen"
  >
    <div class="container">
      <router-view />
    </div>
  </div>
</template>

<script>
import { provide, ref } from 'vue';
import NavbarComponent from './components/NavbarComponent.vue';
export default {
  components: {
    NavbarComponent,
  },
  setup() {
    let isAuthenticated = ref(
      typeof sessionStorage.getItem('user') == 'string'
    );
    provide('authenticated', isAuthenticated);
    return {
      isAuthenticated,
    };
  },
};
</script>
```

Слика 2: Главна компонента App.vue

```
import { createApp } from 'vue';
import App from './App.vue';
import router from './router';
import './assets/tailwind.css';
import './registerServiceWorker'

router.beforeEach((to, from, next) => {
  if (to.matched.some((route) => route.name == 'loginView' || route.name == 'registerView')) {
      const user = JSON.parse(sessionStorage.getItem('user'));
      if(user == null) {
        return next();
      }
      return next('/');
  }
  return next();
});

createApp(App).use(router).mount('#app');
```

Слика 3: main.js - Креирање на Vue апликација

Во нашиот случај ако нема корисник во sessionStorage тогаш то поставуваме isAuthenticated на false при што се знае дека нема логирано корисник и се користи provide(key, value) наредбата.

Во router.beforeEach на слика 3 проверуваме дали сакаме да се прикаже компонентата со име loginView или registerView притоа ако има веќе логирано користни тој ке биде во sessionStorage и ако веке постои таков корисник тогаш ке не прати на Home страната, во спротивно ке продолжи кон Login или регистер формата.

2.3 Компоненти

Компоненти во Vue овозможуваат да се подели една веб страна на повеќе компоненти како на пример имаме компонента за навигација, компонента за пост, компонента за коментар итн, и понатаму во едно View се поврзуваат сите компоненти и се гради страната ова е олеснување на тоа што кога градиме една веб страна ние мора да почнеме header некој па потоа навигација, но кога имаме компоненти ние може почнеме дури и од компонентата за навигација и тоа нема да биде проблем бидејќи ке имаме различни компоненти за било кој дел од веб страната исто така компонентите може да се прекористуваат, компнетите во vue имаат екстензија .vue една компонента се состои од <template> таг во кој се пишува

html и <script> таг во кој се чува vue објектот исто така има и <style> таг кој може да се занемари ако се користи некој CSS framework.

2.4 Composition API

Ако имаме некоја променлива која се наоѓа во Root компонентата и сакаме да ја пристапиме во некоја дете компонента на таа Root компонента тогаш еден начин е тоа да го направиме преку props и така во секоја компонента од Root надолу се додека не стигнеме до таа дете компонента или друг начин е тоа да се направи со Vuex store а трет начин е да се користи Vue Composition API кој е интегриран во Vue 3. Во оваа апликација се користи третиот начин.

На слика 4 е прикажан пример на користење на composition API тука е важен делот setup(), со ref({username: ‘’, password:’’}) се омозможува со што се креира референца од објектот кој потоа преку функцијата provide(key: string, value: obj) се чува за да може да се користи и во следната компонента.

```vue
<template>
  <div class="flex justify-center">
    <div
      class="w-full md:w-3/4 bg-neutral-50 dark:bg-neutral-700 dark:text-white rounded shadow-md p-10"
    >
      <form>
        <fieldset class="border border-solid border-gray-300 p-4 shadow-md">
          <legend class="text-2xl font-extralight">Login</legend>

          <div class="p-6 flex flex-col gap-5">
            <input-component which="user" property="username"></input-component>
            <input-component
              which="user"
              property="password"
              type="password"
            ></input-component>
            <div class="flex justify-center">
              <login-user-button
                class="border-2 border-slate-400 rounded-full p-2 px-5 hover:text-white hover:bg-blue-400 hover:border-blue-400 shadow:md hover:shadow-xl"
              ></login-user-button>
            </div>
          </div>
        </fieldset>
      </form>
    </div>
  </div>
</template>

<script>
import InputComponent from '../components/InputComponent.vue';
import { ref, provide } from 'vue';
import LoginUserButton from '../components/LoginUserButton.vue';
export default {
  name: 'loginView',
  setup() {
    let obj = ref({
      username: '',
      password: '',
    });

    provide('userData', obj);
    return {
      obj,
    };
  },
  components: {
    InputComponent,
    LoginUserButton,
  },
};
</script>
```

Слика 4: LoginView - Composition API

Во Input Component се зема тој објект со функцијата inject(key: string) и бидејќи е референца ако ја смениме вредноста на некој од атрибутите автоматски се менува во секоја компонента (реактивен).

```vue
<template>
  <label class="flex flex-col gap-2">
    <span class="font-bold">{{ label }}</span>
    <input
      :type="type"
      class="shadow-md dark:bg-neutral-600 dark:text-white form-input mt-1 block w-full rounded border"
      :name="obj[property]"
      v-model="obj[property]"
    />
  </label>
</template>

<script>
import { inject } from 'vue';
import { label } from './utility';

export default {
  props: {
    which: String,
    property: String,
    type: { type: String, default: 'text' },
  },
  setup(props) {
    const obj = inject(`${props.which}Data`);
    return {
      obj,
    };
  },
  computed: {
    label() {
      return label(this.property);
    },
  },
};
</script>
```

Слика 5: InsertComponent.vue

3. Земање на податоци од сервер

За да се земат податоци од серверот се користи fetch API кое е веќе вградено во browser и со кое може лесно да се пушти барање до серверот за да се земат податоците и да се прикажат на клиентот барањето се пушта до линк до серверот на и преку и притоа се пушта и GraphQL query со кое што преку Resolver се земаат податоците и се пуштаат во мрежата, а податоците ги добиваме во JSON формат и мора да искористиме res.json() кое парсира json објект кој има дата атрибут во кој се чуваат податоците.

```
this.posts = await fetch('http://localhost:3000/graphql', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    Accept: 'application/json',
  },
  body: JSON.stringify({
    query: `
    query LIST_POSTS {
        posts {
            id
            title
            str
            createdAt
            user {
                id
                username
            }
            comments {
                id
                text
                createdAt
                user {
                    id
                    username
                }
            }
        }
    }
`,
  }),
})
  .then((res) => res.json())
  .then((res) => res.data.posts);
```

Слика 6: fetch request

За да прикачиме слика со GraphQL на сервер исто така се користи fetch со таа разлика што мора да се користи FormData објектот во JavaScript а во FormData објектот со помош на функцијата append се додаваат вредности кој го содржат body-то на request-от притоа во ‘operations’ се поставува mutation и variables објект со кој се пуштаат променливите на сервер, во ‘map’ се поставуваат по реден број елементите од variables објектот кој се пушта во variables објектот и во ‘0’ се пушта самата слика која сакаме да ја прикачиме на сервер и на крај се користи fetch API то за пуштање на барање то сервер.

```
methods: {
  async addImage() {
    if(this.community.admin.id != JSON.parse(sessionStorage.getItem('user')).id) {
      return;
    }
    const input = document.createElement('input');
    const formData = new FormData();
    input.type = 'file';
    input.click();
    input.addEventListener('change', async () => {
      if (input.files.length == 0) {
        return;
      }
      formData.append(
        'operations',
        JSON.stringify({
          query: `
          mutation UPLOAD_AVATAR_COMMUNITY($communityId: String!, $file: Upload!) {
            changeAvatar(file: $file, communityId: $communityId) {
              id
              avatarImage
            }
          }`,
          variables: {
            communityId: this.$route.params.id,
            file: null,
          },
        })
      );
      formData.append('map', '{ "0": ["variables.file"] }');
      formData.append('0', input.files[0]);

      const updatedCommunityAvatar = await fetch(
        'http://localhost:3000/graphql',
        {
          method: 'POST',
          body: formData,
        }
      )
        .then((res) => res.json())
        .then((res) => res.data.changeAvatar);
      this.community.avatarImage = updatedCommunityAvatar.avatarImage;
    });
  },
},
```

Слика 7: fetch multipart data GraphQL image Upload

4. Tailwind

Tailwind е CSS framework кој се користи за стилизирање на оваа веб апликација, со помош на Tailwind може монгу лесно и брзо да се направи стилот на една веб апликација за да се конфигурира Tailwind во една веб апликација се користи tailwind.config.js во кој се пишуваат конфигурации на примерот на слика 8 darkMode значи дека ке се користи преку class атрибутот на веб апликацијата а во theme се стават сите правила кои потоа може да се користат во class атрибутот.

```
module.exports = {
  content: ["./public/index.html",    "./src/**/*.{vue,js,ts,jsx,tsx}"],
  presets: [],
  darkMode: 'class', // or 'class'
  theme: {
    screens: {
      sm: '640px',
      md: '768px',
      lg: '1024px',
      xl: '1280px',
      '2xl': '1536px',
    },
    colors: ({ colors }) => ({
      inherit: colors.inherit,
      current: colors.current,
      transparent: colors.transparent,
      black: colors.black,
      white: colors.white,
      slate: colors.slate,
      gray: colors.gray,
      zinc: colors.zinc,
      neutral: colors.neutral,
      stone: colors.stone,
      red: colors.red,
      orange: colors.orange,
      amber: colors.amber,
      yellow: colors.yellow,
      lime: colors.lime,
      green: colors.green,
      emerald: colors.emerald,
      teal: colors.teal,
      cyan: colors.cyan,
      sky: colors.sky,
      blue: colors.blue,
      indigo: colors.indigo,
      violet: colors.violet,
      purple: colors.purple,
      fuchsia: colors.fuchsia,
      pink: colors.pink,
      rose: colors.rose,
    }),
```

Слика 8: tailwind.config.js

Со dark:bg-sky-800 му кажуваме на веб прелистувачот кој стил да го употреби кога некој корисник сака да ја гледа страната во dark mode а во Navbar-от имаме копче кое го овозможува тоа, исто така може да се користат и превдо кални како што е hover: и со ова може многу лесно да се стилизира еден HTML елемент. Функцијата themeToggle() прикажана на слика 9 при клик на копчето во Navbar му додава на сите елементи во DOM-от класа dark при што сите класи со dark: стапуваат во моќ.

```
methods: {
  themeToggle() {
    if (localStorage.theme === "light") {
      localStorage.theme = "dark";
      document.documentElement.classList.add("dark");
    } else {
      document.documentElement.classList.remove("dark");
      localStorage.theme = "light";
    }
  },
},
created() {
  if (localStorage.theme === undefined) {
    localStorage.theme = "light";
    document.documentElement.classList.remove("dark");
  }
},
```

Слика 10: themeToggle() NavbarComponent.vue

```
<button
  @keyup.enter="submit"
  class="
    text-md
    rounded-full
    border
    shadow-md
    border-blue-400
    bg-white
    px-5
    py-1
    transition-all
    hover:bg-blue-400 hover:text-white hover:dark:text-white
    dark:bg-sky-800 dark:border-0
    hover:dark:bg-sky-700
  "
>
  Comment
</button>
```

Слика 9: Пример button со tailwind калси

5. Server

За backend делот се користи TypeScript и сервер со node.js apollo-server-express заедно со express со app.use() се поставува middleware кој се извршува пред секое барање до сервер, subscriptions-transport-ws користи за сервер за субскрипции за кога ќе се внеси нешто во базата итн и сите работи одат преку GraphQL.

```
import { UserResolver } from './resolvers/UserResolver';
import { CommunityResolver } from './resolvers/CommunityResolver';
import { PostResolver } from './resolvers/PostResolver';
import { CommentResolver } from './resolvers/CommentResolver';
import { CountryResolver } from './resolvers/CountryResolver';
import { CategoryResolver } from './resolvers/CategoryResolver';
import { buildSchema, buildTypeDefsAndResolvers } from 'type-graphql';
import { Container } from 'typedi';
import UserModel from '../models/UserModel';
import CommunityModel from '../models/CommunityModel';
import PostModel from '../models/PostModel';
import CommentModel from '../models/CommentModel';
import CountryModel from '../models/CountryModel';
import CategoryModel from '../models/CategoryModel';


Container.set({ id: 'USER', factory: () => UserModel });
Container.set({
  id: 'COMMUNITY',
  factory: () => CommunityModel,
});
Container.set({ id: 'POST', factory: () => PostModel });
Container.set({
  id: 'COMMENT',
  factory: () => CommentModel,
});
Container.set({
  id: 'COUNTRY',
  factory: () => CountryModel,
});
Container.set({
  id: 'CATEGORY',
  factory: () => CategoryModel,
});

export const schema = buildSchema({
  resolvers: [
    UserResolver,
    CommunityResolver,
    PostResolver,
    CommentResolver,
    CountryResolver,
    CategoryResolver,
  ],
  emitSchemaFile: true,
  container: Container,
});

export const typeDefsAndResolvers = buildTypeDefsAndResolvers({
  resolvers: [
    UserResolver,
    CommunityResolver,
    PostResolver,
    CommentResolver,
    CountryResolver,
    CategoryResolver,
  ],
  container: Container,
});
```

Слика 11: buildSchema.ts GraphQL, DI

- Dependency Injection

Се користи Container од typedi што е библиотека за typescript за Dependency Injection на почеток се сетираат сите модели кои се експортирани од Mongoose а потоа тој таа промелива Container се праќа во објектот кој се праќа на buildSchema која е направена од GraphQL моделите.

```
import 'reflect-metadata';
import express from 'express';
import Mongoose from 'mongoose';
import bodyParser from 'body-parser';
import path from 'path';
import cors from 'cors';
import { graphqlUploadExpress } from 'graphql-upload';
import { createServer } from 'http';
import { SubscriptionServer } from 'subscriptions-transport-ws';
import { subscribe, execute } from 'graphql';
import { ApolloServer } from 'apollo-server-express';

import { schema, typeDefsAndResolvers } from './graphql/schema';

async function startServer() {
  const PORT = process.env.PORT || 3000;
  const app = express();
  app.use(graphqlUploadExpress({ maxFileSize: 10000000, maxFiles: 10 }));
  app.use(express.static(path.join(__dirname, 'public')));
  app.use(bodyParser.json());
  app.use(cors());

  const ws = createServer(app);
  const sch = await schema;
  const subscriptionServer = SubscriptionServer.create(
    {
      execute,
      subscribe,
      schema: sch,
      onConnect() {
        console.log('connected');
      },
      onDisconnect() {
        console.log('disconnect');
      },
    },
    { server: ws, path: '/graphql' }
  );

  const schTypeDefsAndResolvers = await typeDefsAndResolvers;
  const apollo = new ApolloServer({
    typeDefs: schTypeDefsAndResolvers.typeDefs,
    resolvers: schTypeDefsAndResolvers.resolvers,
  });

  app.use('/graphql', bodyParser.json());

  await Mongoose.connect('mongodb://localhost:27017/');
  await apollo.start();
  apollo.applyMiddleware({ app });
  ws.listen(PORT, () => {
    console.log(
      `GraphQL running on http://localhost:${PORT}${apollo.graphqlPath}`
    );
  });
}

startServer();
```

Слика 12: server.ts

5.1 GraphQL

Оваа апликација користи GraphQL кој овозможува да се направи API кое е брзо и флексибилно во GraphQL има шеми кој кажуваат какви атрибути има некој модел и ресолвери кои ги користат моделите за да направи некое query, mutation или subscription.

Бидејќи работиме во TypeScript ќе ја користиме библиотеката TypeGraphQL која работи со анотации и овозможува лесно да се направи модел или ресолвер.

- GraphQL Resolver

На слика 13 е прикажан еден пример на ресолвер од оваа апликација кој е анотиран со анотацијата @Resolver() која му покажува на шемата дека е Resolver и @Service() кој му кажува на Container-от од претходно дека ова е Service() и оваа анотација автоматски го додава овој Resolver во Container, во конструкторот на оваа класа се инјектираат сервисите communityService и userService бидејќи за да креираме community негде не треба User-от кој го креира тоа community ако сакаме да вратиме некое query или листа елементи тогаш ја користиме анотацијата @Query(returns => [type]) која кажува каков тип треба да врати функцијата со @Arg() се означуваат аргументите кој треба да ги прими функцијата, а ако сакаме да направиме мутација за додавање на елемент во база ја користиме анотацијата @Mutation(arg) која исто така прима функција како Query што кажува каков тип ке враќа, тука се користи и библиотеката graphql-upload која овозможува upload на некој file на сервер како функцијата changeAvatar() која прима аргументи file кој е од тип GraphQLUpload и communityId за да знае за кое community се однесува сликата која ке биде пратена од корисникот.

```
import { FileUpload, GraphQLUpload } from 'graphql-upload';
import { Arg, Mutation, Query, Resolver } from 'type-graphql';
import { Service } from 'typedi';
import { CommunityService } from '../../services/CommunityService';
import UserService from '../../services/UserService';
import { Community, CommunityInput } from '../schema/CommunitySchema';

@Service()
@Resolver()
export class CommunityResolver {
  constructor(
    private readonly communityService: CommunityService,
    private readonly userService: UserService
  ) {}

  @Mutation((returns) => Community)
  async changeAvatar(
    @Arg('communityId') communityId: string,
    @Arg('file', () => GraphQLUpload) file: FileUpload
  ) {
    return await this.communityService.updateAvatar(communityId, file);
  }

  @Query((returns) => [Community])
  async communities() {
    return await this.communityService.findAll();
  }

  @Query((returns) => Community)
  async community(@Arg('communityId') communityId: string) {
    return await this.communityService.findById(communityId);
  }

  @Mutation((returns) => Community)
  async createCommunity(
    @Arg('categoryName') categoryName: string,
    @Arg('adminId') adminId: string,
    @Arg('communityInput') communityInput: CommunityInput
  ) {
    return this.communityService.create(categoryName, adminId, communityInput);
  }

  @Mutation((returns) => Community)
  async deleteById(@Arg('communityId') communityId: string) {
    return await this.communityService.deleteById(communityId);
  }

  @Mutation((returns) => Community)
  async joinCommunity(
    @Arg('communityId') communityId: string,
    @Arg('userId') userId: string
  ) {
    return this.communityService.joinCommunity(communityId, userId);
  }
}
```

Слика 13: GraphQL Resolver - CommunityResolver.ts

- GraphQL Object Schema

За да се направи шема за некој објект се користи класа која се анотира со анотацијата @ObjectType() кој му покажува на TypeGraphQL-от дека е шема за некој објект анотацијата @Field(is => ID) покажува дека id атрибутот ке биде за идентификување на објектот од кога ке се земе во базата а само @Field() покажува да се земе во шемата на овој објект ако некој атрибут немање анотација @Field() тогаш кога ке направевме барање до сервер тој атрибут немаше да можиме да го земиме.

```
import { Field, ObjectType, ID, InputType } from 'type-graphql';
import { Post } from './PostSchema';
import { User } from './UserSchema';

@ObjectType()
export class Community {
  @Field((is) => ID)
  id!: String;

  @Field()
  name!: String;

  @Field()
  description!: String;

  @Field((is) => [Post])
  posts!: Post[];

  @Field((is) => User)
  admin!: User;

  @Field((is) => [User])
  moderators!: User[];

  @Field({ nullable: true })
  avatarImage!: String;
}

@InputType()
export class CommunityInput {
  @Field()
  name!: String;

  @Field()
  description!: String;
}
```

Слика 14: GraphQL Schema CommunityModel.ts

Сите примитивни типови како што се String, Date итн се земаат автоматски а сите други типови кој се креирани од корисник мора да внесат како @Field(is => User) бидејќи овие типови неможе автоматски да се знаат исто така @Field() прима и објект како аргумент кој служи за опции на тој атрибут за тоа дали тој може да биде null и др.

Со @InputType() се анотира класата која ке може да се користи после како аргумент во некоја функција во ресолверот од GraphQL-от.

- GraphQL Subscription

Анотацијата @Subscription() овозможува кога ке се стави некој објект преку некоја мутација да се случи event преку pubSub.publish(key: string, value: Obj) да се извршат сите субскрипции кој имаат Subscription со topics: 'POST_ADDED) исто така @PubSub() анотацијата служи за да се инјектира pubSub што значи publish / subscribe.

```
@Mutation((returns) => Post)
async createPost(
  @Arg('userId') userId: string,
  @Arg('communityId') communityId: string,
  @Arg('postInput') postInput: PostInput,
  @PubSub() pubSub: PubSubEngine
) {
  let post = await this.postService.create(userId, communityId, postInput);
  await pubSub
    .publish('POST_ADDED', post)
    .then(() => console.log('published'));
  return post;
}

@Subscription((returns) => Post, { topics: 'POST_ADDED' })
async postAdded(@Root() payload: string) {
  console.log(payload);
  return payload;
}
```

Слика 15: GraphQL subscription notification - PostResolver.ts

5.2 Mongoose

Во оваа апликација за база MongoDB а mongoose Object Data Modelling (ODM) библиотека за Node.js преку која се прават моделите за базата на податоци.

За да се направи модел преку Mongoose во TypeScript прво мора да се направи interface за какви атрибути ке има тој објект какви типови ке имаат тие атрибути, тука е прикажан interface за моделот User.

```ts
import Mongoose from 'mongoose';
import { ICommunity } from './CommunityModel';
import { IPost } from './PostModel';
import { IComment } from './CommentModel';

export enum Gender {
  Man = 'Man',
  Woman = 'Woman',
  Other = 'Other',
}

export interface IUserOptions {
  inboxMessages: boolean;
  upvotesOnComments: boolean;
  upvotesOnPosts: boolean;
  newFollowers: boolean;
}

export interface IUser {
  username: string;
  password: string;
  firstName: string;
  lastName: string;
  email: string;
  birthDate: Date;
  gender: Gender;
  location: string;
  communities: ICommunity[];
  posts: IPost[];
  comments: IComment[];
  createdAt: Date;
  options: IUserOptions;
}
```

Слика 16: Mongoose Schema Interfaces

Потоа се креира шема за секој interface со неговите атрибути. type покажува каков тип ке има атрибутот и type атрибутот исто така може да биде и друга шема во објектот на шемата немора да се кажува атрибут за ID бидејки Mongoose автоматски го генерира тоа и може да има други опции како што е timestamps што креира атрибути за кога бил креиран објект од тип User како createdAt и updatedAt.

```ts
export const UserOptionsSchema: Mongoose.Schema = new Mongoose.Schema({
  inboxMessages: { type: Boolean },
  upvotesOnComments: { type: Boolean },
  upvotesOnPosts: { type: Boolean },
  newFollowers: { type: Boolean },
});

export const UserSchema: Mongoose.Schema = new Mongoose.Schema(
  {
    username: { type: String, required: true },
    password: { type: String, required: true },
    firstName: { type: String, required: false },
    lastName: { type: String, required: false },
    email: { type: String, required: true },
    birthDate: { type: Date, required: false },
    gender: {
      type: String,
      required: false,
      default: Gender.Other,
      enum: Gender,
    },
    location: { type: String, required: false },
    communities: [
      {
        type: Mongoose.Schema.Types.ObjectId,
        required: false,
        ref: 'Community',
      },
    ],
    posts: [
      { type: Mongoose.Schema.Types.ObjectId, required: false, ref: 'Post' },
    ],
    comments: [
      { type: Mongoose.Schema.Types.ObjectId, required: false, ref: 'Comment' },
    ],
    options: {
      type: UserOptionsSchema,
      required: false,
    },
  },
  { timestamps: true }
);

export default Mongoose.model<IUser & Mongoose.Document>('User', UserSchema);
```

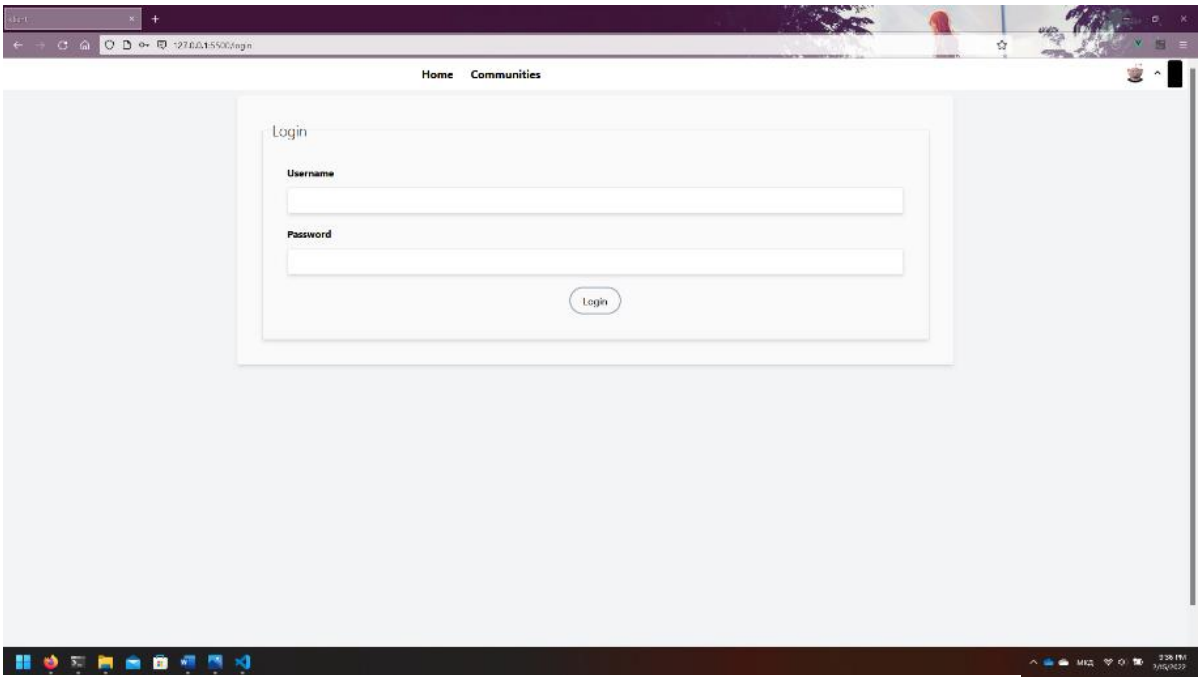
Слика 17: Mongoose Schema - UserModel.ts

6. Главна апликација

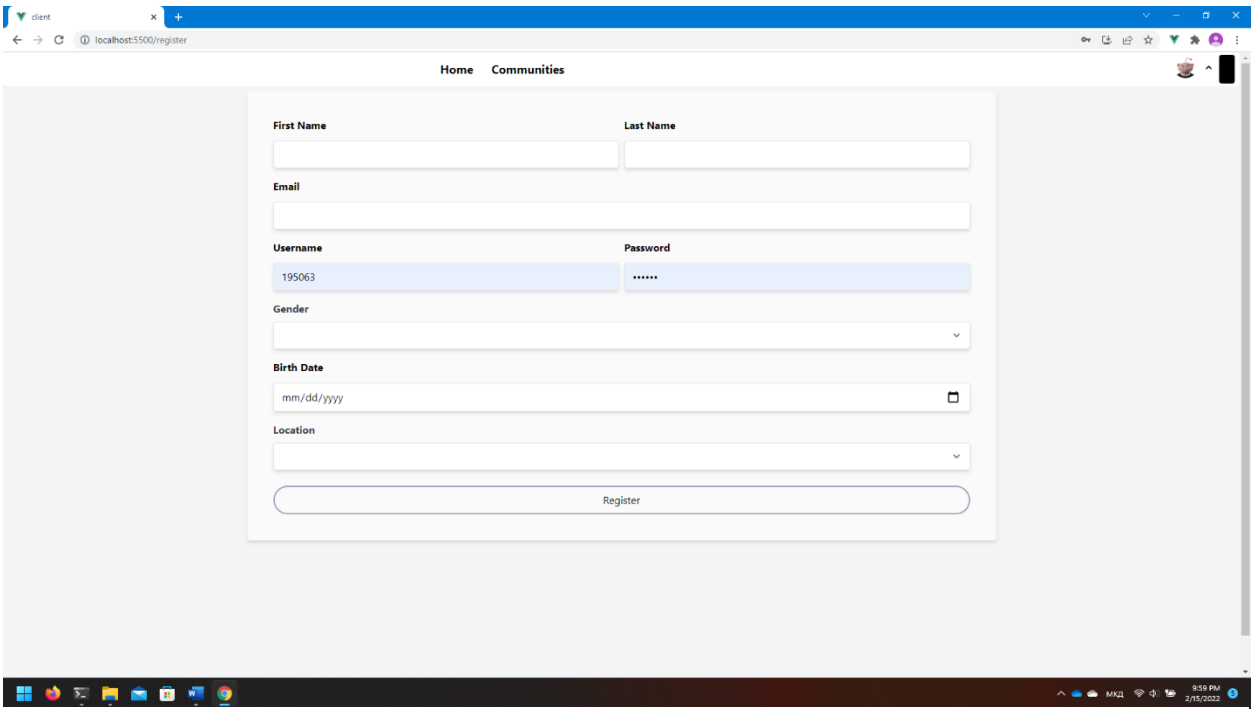
Апликацијата е функционален форум кој има автентикација и може некој корисник да се логира или прво да се регистрира па потоа може да прави негови Communities или пак да влезе во веќе направени communities да постира во нив и исто така да коментира на постови кои се направени од други корисници и исто така може да се инсталира преку Chrome.

6.1 Login и Register форми

Преку формата на слика 18 некој корисник може да се логира со внесување на username и password при клик на копчето Login се праќа Request со fetch API кој ако е пронајден корисникот го враќа ако не е пронајде таков корисник фрла exception на серверот при што серверот враќа порака Invalid username and password, а преку формата за регистрација може некој да стане корисник на овој форум.



Слика 19: Login form

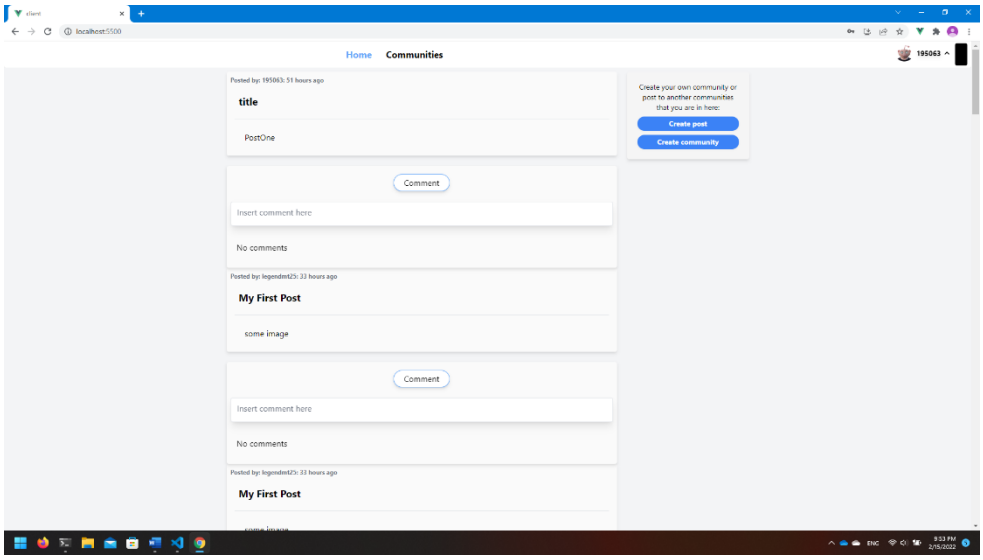


Слика 18: Register form

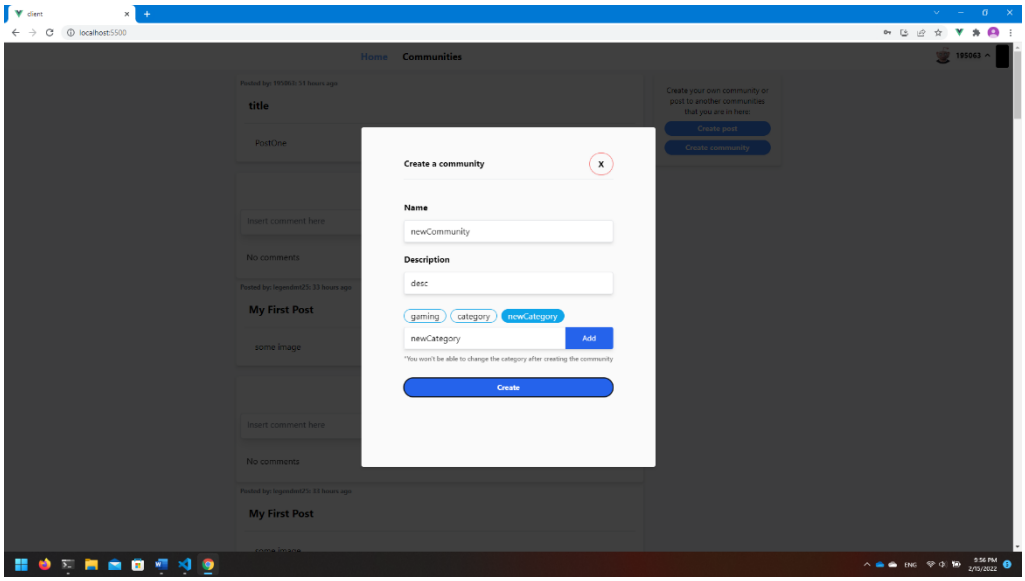
6.2 Home page

После од кога одреден корисник ке се логира го носи на Home страната на која има некој рандом постови од community и на оваа страна има две копчиња, едно за create community кое отвора модал за внесување информации за community и едно

за креирање на пост кое го носи корисникот на друга страна за внесување на информации за да се направи еден пост.

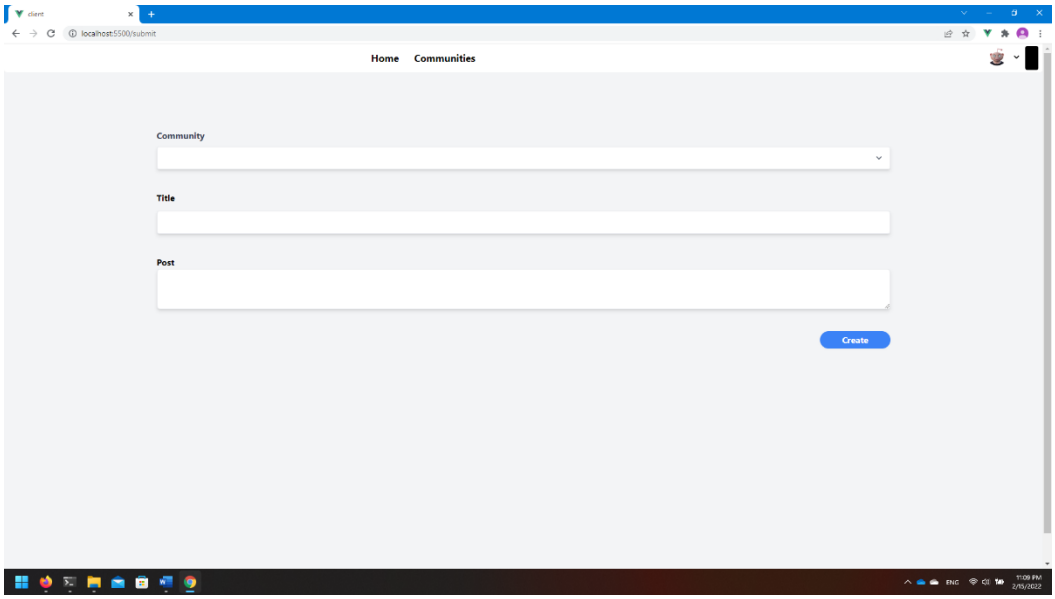


Слика 21: Home page



Слика 20: Insert community modal

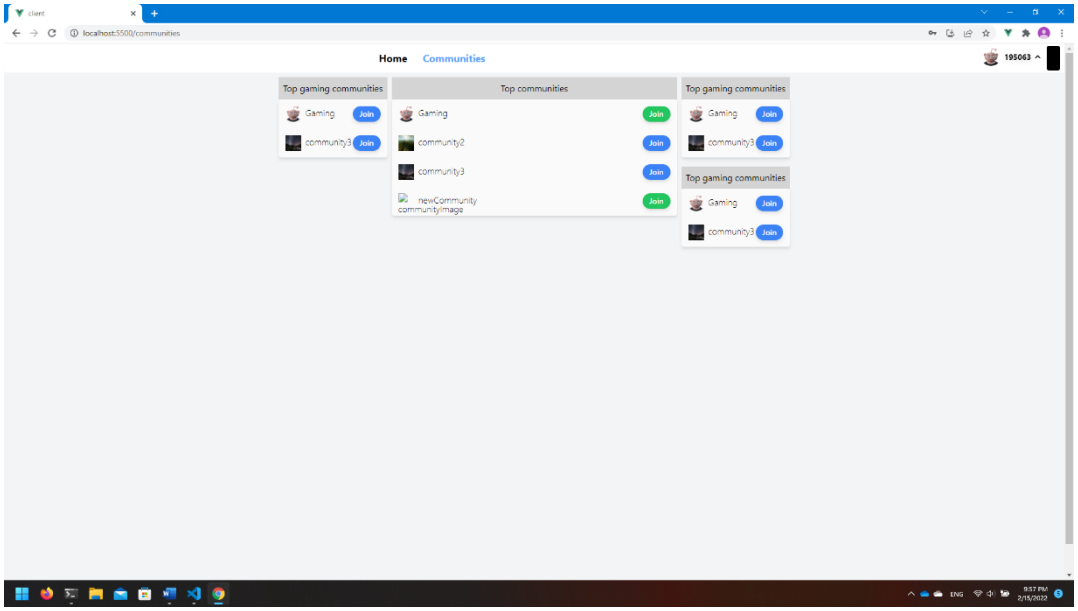
На слика 20 е прикаж модал кој се отвора при клик на копчето Create community тука може да се додаде категоријата на community кое сака корисникот да го направи а копчето Create работи само кога има корисник логирано бидејќи потребни се информации за корисникот за да се постави како админ на Community.



Слика 22: Create Post /submit

Страната на слика 22 се појавува при клик на копчето Create post и тука се внесуваат информациите за креирање на post.

6.3 Communities

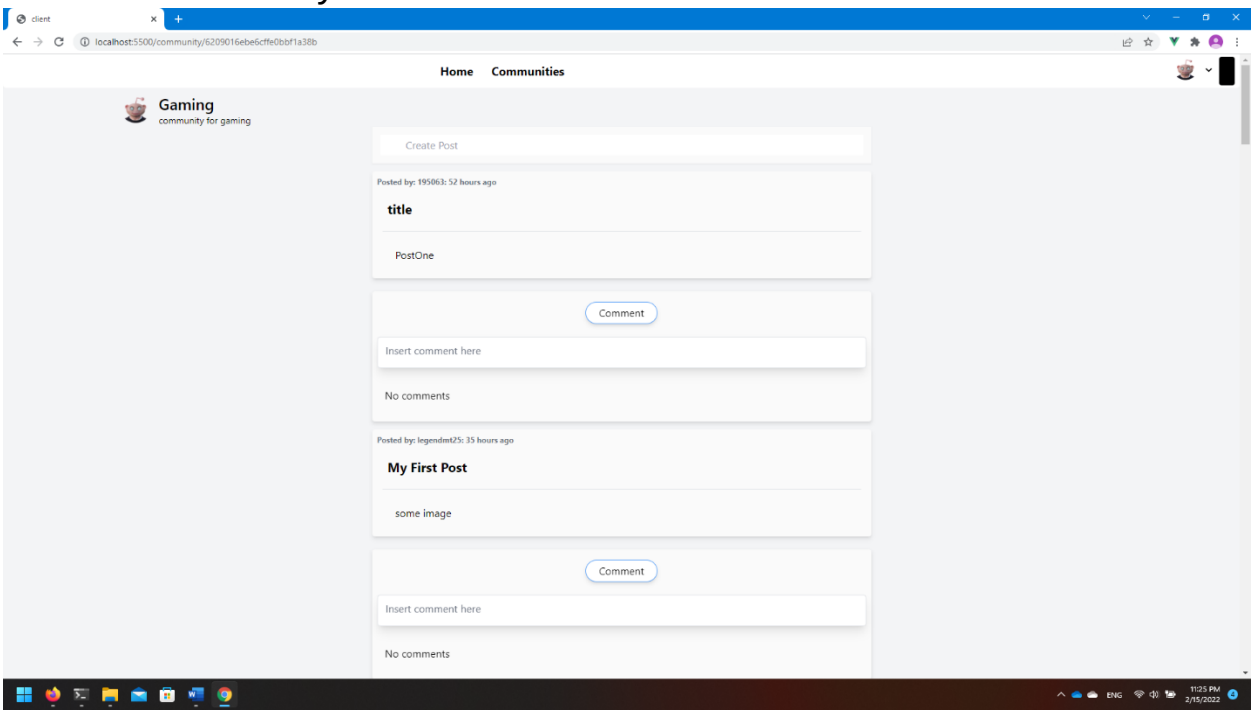


Слика 23: CommunitiesList /communities

На слика 23 е прикажана страната на која се прикажуваат сите communities и од страните некој communities кој се од рандом категорија на сликата тоа е gaming бидејќи за сега нема други communities но ако имаше повеќе категории креирано тогаш за секоја компонента ке се земаат рандом категории и communities на таа рандом категорија. Кога корисникот ке притисне на копшето Join ако успешно бил

внесен во тоа Community тогаш копчето станува зелено. Со притискање на некое Community го редиректира корисникот на друга страна /communities/[id community]

6.4 Community Details



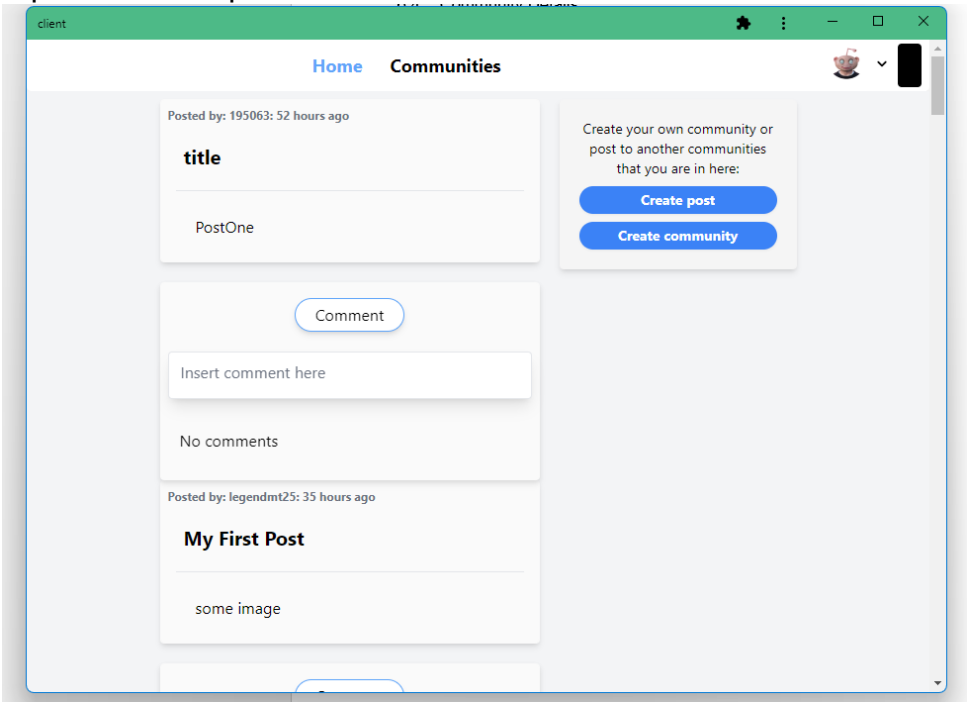
Слика 24: CommunityDetails /community/:id

На слика 24 е прикажана страна која покажува деталји само за едно community, како што се дескрипција за што е, името, сликата како аватар.

При клик на input елементот на Create Post го носи корисникот на рутата /submit, при клик на аватар сликата до gaming му се овозможува на корисникот да ја смени со друга при што се користи fetch со FormData за да се прати прикачи сликата на серверот за да може GraphQL resolver-от да ја земе таа слика и да ја стави на сервер.

7. Инсталација на апликацијата

Оваа апликација може да се инсталира преку Google chrome при клик на копчето горе на URL барот



Слика 25: Инсталирања апликација PWA