Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ДИЗАЙН ТА РОЗРОБКА КОЛОБОРАТИВНОЇ СИСТЕМИ ДЛЯ ОБЛІКУ РОБОЧІХ ДНІВ ТА ЧЕРГУВАНЬ

Текстова частина до курсової роботи
за спеціальністю «Комп'ютерні науки» 122

Керівник курсової роботи

Бабич Т. А. _____

(підпис)

«____» _____ 2021 р.

Виконав студент 4 р.н. БП
«Комп'ютерні науки»
Яцишин І. Є.

«____» _____ 2021 р.

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Керівник курсової роботи
Бабич Т. А.

_____
(підпис)
«____» _____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

Студенту 4-го курсу, Факультету Інформатики
Яцишину Іллі Євгеновичу

ТЕМА Дизайн та розробка колоборативної системи для обліку робочіх днів та чергувань

Зміст ТЧ до магістерської роботи:

Зміст
Анотація
Вступ
1 Огляд прецедентів та передумов створення продукту, формулювання вимог до продукту
2 Огляд реалізації та перевірка рішення
Висновки
Список використаної літератури

Дата видачі «___» _____ 2021 р.

Керівник _____ (підпис)

Завдання отримав _____ (підпис)

Тема: Використання суфіксних дерев для швидкого пошуку у вебі

Календарний план роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1	Отримання завдання на курсову роботу	15.10.2021	
2	Огляд літератури за темою роботи	15.02.2022	
3	Формулювання вимог до додатку, що накладаються предметною областю	15.03.2022	
4	Дослідження існуючих альтернатив	25.03.2022	
5	Реалізація прогрманої частини	01.05.2022	
6	Написання пояснювальної роботи	10.05.2022	
7	Аналіз результатів із науковим керівником	15.05.2022	
9	Коригування роботи за результатами попереднього огляду	17.05.2022	
10	Остаточне оформлення пояснювальної роботи	17.05.2022	

Студент Яцишин І. Є. _____

Керівник Бабич Т. А. _____

«___» _____

Зміст

Анотація

Метою курсової роботи є дизайн та розробка веб додатку – колоборативної системи для ведення обліку робочіх днів та чергування.

Було розглянуто існуючі альтернативи, визначено та реалізовано необхідні частини додатку згідно з вимогами предметної області. Імплементовано процеси реєстрації та авторизації для надання захищенного доступу до інформації та її зміни користувачам системи.

Ключові слова: веб додаток, колаборативна система, календар, розклад.

Вступ

Будь-який бізнес вимагає зручного інструменту для контролю та адміністрування розкладу працівників. Можливість зберігати, швидко й зручно переглядати, та гнучко контролювати робочі зміни кожного окремого учасника команди є невід'ємною частиною будь-якої компанії, що претендує на успіх на ринку. Це стосується майже всіх доменів виробництва та надання послуг, від команд професійних програмістів до пекарів у булочній. Без інструментів для створення та аналізу робочих графіків кожного з команди неможливо планувати робочий процес та відпочинок.

Додатковою умовою стає специфіка деяких команд, учасники яких в тий чи іншій формі займаються чергуванням впродовж робочого дня. Так, наприклад, кожен спеціаліст технічної підтримки може годину на день слідкувати за стабільністю систем, а адміністратори стоматології по черзі приймати дзвінки клієнтів. Облік таких чергувань зазвичай відбувається на бумазі, що сповільнює робочі процеси, зменьшуючи ефективність бізнесу. Окрім того, людський фактор залишае можливість помилки, коли працівник забуває про чергування або плутає черговість. Такі помилки ставлять під загрозу бізнес процеси, оскільки чергування зазвичай з'являється саме через необхідність слідкувати та реагувати на важливі для бізнесу події в режиму реального часу.

Вирішенням перелічених питань може стати єдина синхронізована система обліку, що б зберігала в собі інформацію як про робочі, вихідні, лікарняні дні кожного з працівників, так і про стан чергування.

Окрім цього, автоматизована система може взяти на себе задачу вчасно нагадувати кожному з працівників про необхідність виконати ту чи

іншу дію, зводячи на мінімум можливість помилки з боку учасників команди.

РОЗДІЛ 1: Огляд прецедентів та передумов створення продукту

Реалізований впродовж виконання роботи програмний продукт планується використати для існуючої команди, що потребує можливості зручно контролювати та переглядати розклад її учасників.

Додатковою задачею є реалізація синхронізованного з розкладом компоненту, що б відповідав за чергування, надаючи можливість зручно та швидко створювати розклад чергування, та нагадуючи черговим про необхідність вчасно приступити до своїх обов’язків. Під синхронізацією мається на увазі автоматичний підрахунок та призначення чергових, що б базувався на переліку працівників, присутніх в день проведення чергування.

Існує також вимога зберігати визначену кількість попередніх розкладів та надавати можливість зручно повертатися до них, враховуючи новий розклад та різницю в переліку присутніх працівників на момент створення черги.

Окремою вимогою є можливість динамічного додавання нових команд з унікальними налаштуваннями, робочі процеси яких відрізняються від інших команд, що використовують додаток. Команди мають бути відокремленими та такими, що не впливають на розклад або чергування одна одної.

Ще одним важливим питанням є захищенність системи, реалізація процесу реєстрації та подальше отримання авторизованого доступу до чутливої інформації про користувачів компанії та її внутрішніх процесів.

Окрім цього, необхідна реалізація можливості змінювати поведінку додатку відповідно до вимог команди. Наприклад, такими змінами може стати скорочення робочого дня, зміна часу початку робочого дня, зміна

вихідних днів (наприклад, команда вирішила відпочивати в понеділок замість суботи), або нові типи робочіх змін та вихідних (наприклад, «нічна зміна», «неоплатна відпустка», «декрет» тощо).

Нарешті, цікавим доповненням може стати оновлення інформації для кожного з працівників в режимі реального часу, без необхідності оновлювати сторінку для отримання змін. Таким чином, можна пришвидшити та зробити наочніше адміністрування розкладу, надаючи кожному працівнику можливість спостерігати за діями своїх колег одразу після їх здійсення.

РОЗДІЛ 2: Огляд реалізації та перевірка рішення

2.1 Огляд обраних технологій та сервісів для створення додатку

Для створення веб застосунку було обрано стак MERN, що вимагає використання наступних технологій:

1. Node.js в якості середи виконання мови JavaScript на стороні сервера
2. Бібліотеку Express для спрощення реалізації REST API, покликаного обслуговувати комунікацію між клієнтом та сервером
3. Документарну базу даних MongoDB для збереження інформації про команди, їх коніфгурації, користувачів, розклади та чергування
4. Фреймворк React для швидкої побудови зручного інтерфейсу на стороні клієнта.

Додатково до перелічених технологій використовуються наступні:

1. Slack API – API найбільш популярного робочого месенджеру Slack, що надає можливість отримувати доступ до робочих чатів користувачів та самостійно публікувати нагадування про чергування кожному з учасників команди
2. Docker та Docker Compose – технології для контейнеризації та подальшого розгортання додатку на справжнії серверах. Docker Compose відіграє роль зручного інструменту для реалізації зв’язку та комунікації між контейнерами
3. Nginx – веб сервер, що в застосунку використовується для реалізації проксіфікації під час взаємодії з React додатком та Node.js сервером

4. Npm – менеджер пакетів для Node.js для встановлення та адміністрування використаних бібліотек та фреймворків (в тому числі React, Express, Nginx)
5. Mongoose – бібліотека, що реалізує з'єднання між Node.js сервером та MongoDB базою даних. Надає поняття схеми (моделі) для оперування сутностями, що зберігаються в базі даних (команди, клієнти тощо).

2.2 Огляд компонентів застосунку та способів їх взаємодії

Центральною частиною застосунку є Node.js сервер, доступ до якого React клієнт отримує через проксі Nginx (рисунок 2.1).

client > nginx.conf

```
server {
    listen 80;
    location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
        try_files $uri $uri/ /index.html =404;
    }
    location /api {
        proxy_pass http://server:3001;
    }
    location /api/update {
        proxy_pass http://server:3001;
        proxy_http_version 1.1;
        proxy_set_header Connection "";
    }
}
```

Рисунок 2.1 – Конфігурація Nginx

Конфігурація демонструє правила, за якими запити від клієнту переспрямовуються до серверу з відповідною конфігурацією в залежності

від отриманого шляху. Так, наприклад, всі запити зі шляхом, що починається з “/api” будуть обслуговуватися сервером, а без нього – власне React’ом. Винятком є запити зі шляхом “api/update”, що слугують для встановлення SSE (Server-Sent Events) з’єднання між сервером та клієнтом. Це потрібно для реалізації оновлення в режимі реального часу, описаного в розділі 1. Такі запити мають уникальні хедери.

Сервер, клієнт та база даних між собою об’єднані в Docker мережу, що надає можливість комунікувати безпосередньо всередені середи Docker (рисунок 2.2).

```
services:
  client:
    image: registry.hrzn.io/support/wg-client
    stdin_open: true
    ports:
      - "3000:80"
    depends_on: …
    networks:
      - watch-guard
  server:
    image: registry.hrzn.io/support/wg-server
    ports:
      - "3001:3001"
    depends_on: …
    networks:
      - watch-guard
  mongo:
    image: mongo
    ports:
      - "27018:27017"
    networks:
      - watch-guard
    volumes: …
    environment: …
networks:
  watch-guard:
    driver: bridge
```

Рисунок 2.2 – Конфігурація Docker Network

Кожен з компонентів є частиною мережі “watch-guard”, а одже може комунікувати з іншими за допомогою відкритів портів, вказаних як значення властивостей “ports”. Це необхідно зокрема і для реалізації Nginx проксі, згаданого вище. Окрему роль мережа відіграє для зв’язку між сервером та базою даних, дозволяючи використовувати стандартний

Connection String для отримання доступу до бази розгорнутої в середі Docker.

Клієнт, в свою чергу, є більш відокремленою частиною, що також знаходиться в одному з контейнерів та комунікує лише з сервером.

2.3 Огляд структури бази даних

Дані зберігаються в чьотирьох документарних колекціях (calendars, schedules, teams, users) для зберігання розкладів, чергувань, команд та користувачів відповідно.

Між собою сутності пов’язані наступним чином:

1. Кожен документ з колекцій users, calendars та schedules пов’язані з одним документом з колекції teams (зв’язок один до багатьох) за допомогою вказання ідентифікатору команди як одне з полів користувача
2. Кожен документ з колекції calendars та schedules містить в собі одне або жодного згадування користувача за допомогою вказання його ідентифікатору як елементу масиву (для збереження черговості, рисунок 2.3) або ключа (для присвоєння однієї з можливих типів змін)

Рисунок 2.3 – Зв’язок між розкладом та черговими

Кожен документ зберігає в собі важливу для бізнес-логіки інформацію про сутність, як, наприклад, ім’я та роль користувача в системі, або назву та конфігурацію команди (рисунок 2.4)

Рисунок 2.4 – Сутність користувача представлена в MongoDB

2.4 Огляд архітектури серверу

Сервер складається з менеджерів шляхів, контролерів, обробників та «двигунів». Додаткові частини включають в себе механізми безпеки та авторизації-автентифікації (middleware), валідатори, менеджера Slack боту, нестандартні помилки та Mongoose схеми.

Менеджери шляхів, котролери та обробники відповідають за обробку клієнтських запитів.

Двигуни необхідні для автоматизації процесів створення нових розкладів (одразу пілся закінчення попереніх), нормалізації цих розкладів відповідно до присутніх працівників, та планування розсилки повідомлень в Slack.

Механізми безпеки відповідають за авторизацію та автентифікацію користувачів, встановлення їх належності до команди та перевірку на права на виконання запитів.

Нестандартні помилки виконують роль носіїв інформації про інцеденти, що відбуваються під час виконання запиту, із метою коректної та очікуваної відповіді користувачу в разі появи якоїсь з помилок.

Менеджер Slack боту відповідає за надсилання повідомлень через Slack API.

Mongoose схеми покликані представляти сутності з бази даних як моделі всередині додатку.

2.5 Огляд процесу обробки запитів

Шлях кожного запиту починається з модуля Server, задача якого полягає в перенаправленні запиту до відповідного менеджеру шляхів (рисунок 2.5).

```
const express = require("express");

const signinRouter = require("./routes/signin");
const signupRouter = require("./routes/signup");
const teamRouter = require("./routes/team");
const userRouter = require("./routes/user");
const scheduleRouter = require("./routes/schedule");
const calendarRouter = require("./routes/calendar");
const updateRouter = require("./routes/update");
const probeRouter = require("./routes/probe");

const app = express();

app.use(express.urlencoded({ extended: true }));
app.use(express.json());

app.use("/api/signin", signinRouter);
app.use("/api/signup", signupRouter);
app.use("/api/team", teamRouter);
app.use("/api/user", userRouter);
app.use("/api/schedule", scheduleRouter);
app.use("/api/calendar", calendarRouter);
app.use("/api/update", updateRouter);
app.use("/api/probe", probeRouter);

function launch(port) {
    app.listen(port);
}

module.exports = {
    launch
}
```

Рисунок 2.5 – Модуль Server

Менеджер шляху, в свою чергу, відповідає за подальший аналіз та передання запиту відповідному контроллеру. Він також накладає необхідні для кожного запиту проміжні обробники, що відповідають за авторизацію та автентифікацію запиту (рисунок 2.6).

```
7   router.get(
8       "/",
9       middleware.authenticateToken,
10      middleware.authorizeUser,
11      middleware.authorizeTeam,
12      teamController.get
13  );
14
15  router.get(
16      "/all",
17      teamController.getAll
18  );
19
20  router.patch(
21      "/",
22      middleware.authenticateToken,
23      middleware.authorizeUser,
24      middleware.authorizeTeam,
25      middleware.authorizeAdmin,
26      teamController.patch
27  );
```

Рисунок 2.6 – Менеджер шляхів накладає проміжні обробники та передає запит до контроллера

Рисунок 2.6 демонструє, як запити різного рівня доступу обробляються на цьому етапі. Так, наприклад, отримання всіх команд “GET /all” не вимагає жодної автентифікації, оскільки потрібне ще на моменті реєстрації. В свою чергу, для отримання конкретної (своєї) команди “GET /” користувачу необхідно мати токен, раніше згенерований та виданий сервером. Окрім перевірки того, що токен було підписано саме цим сервером, робиться перевірка на існування користувача, а потім й на його належність до якоїсь з команд. Нарешті, зміна команди “PATCH /” окрім всіх згаданих перевірок потребує також перевірку ролі користувача (auhorizeAdmin) для визначення, чи дозволено йому робити зміни в конфігурації команди.

Нарешті, після проходження всіх перевірок запит потрапляє до контролера, задача якого полягає в передачі запиту до кінцевого обробника та опрацюванні помилок, що можуть виникнути під час обробки (рисунок 2.7).

```
24  async function patch(req, res) {
25      try {
26          await teamManager.patch(req.user.team, req.body.patch);
27          res.status(200).json({ message: "team patched successfully" });
28      } catch (error) {
29          console.error(error);
30          res.status(error.status ?? 400).json({ message: error.message ?? "bad request" });
31      }
32  }
```

Рисунок 2.7 – Контроллер передає запит до обробника та обробляє помилки

Можна помітити, що об’єкт error має незвичайні для стандартної помилки властивості, як, наприклад, status. Контроллер оперує з нестандартними помилками, що створені для репрезентації відомої серверу помилки (наприклад, 404 – не знайдено). Якщо відбулася помилка незнайома серверу, користувачу повернеться загальна помилка 400 – помилковий запит.

Кінцевою точкою подорожі для запита стає обробник, що фактично виконує виконує запит та за необхідності генерує відповідь на нього (рисунок 2.8)

```
45  async function patch(_id, patch) {
46      if (!exists(_id)) {
47          throw new errors.NotFound(404, "Team not found");
48      }
49      await Team.findByIdAndUpdate(_id, patch);
50  }
```

Рисунок 2.8 – Обробник перевіряє, чи існує вказана команда, та оновлює її відповідно до запиту

Рисунок 2.8 демонструє примітивний обробник націлений на оновлення даних. Втім, існують більш комплексні обробники що реалізують складнішу бізнес-логіку, як, наприклад, обробник зміни розкладу (рисунок 2.9), що для виконання задачі здійснює декілька запитів до бази даних після чого оновлює інформацію в цільовій колекції. Окрім цього, можна побачити виклик двох івентів – schedule-update та client-update. Вони необхідні для того, щоб дати серверу зрозуміти, що необхідно оновити чергу (якщо якісь люди більше не працюють на момент зміни), та повідомити всіх клієнтів, хто має відкритим додаток про зміни – так досягається оновлення інтерфейсу в режимі реального часу. Пізніше буде продемонстрований процес підписок/відписок на розсилку івентів та оновлення інтерфейсу.

```
async function patch(team, patches) {
    for (patch of patches) {
        patch.date = new Date(patch.date);
        if (!await exists({ date: patch.date, team })) {
            throw new errors.NotFound(404, "Calendar not found");
        }
        await Calendar.findOneAndUpdate(
            {
                date: patch.date,
                team
            },
            {
                $set: {
                    ...Object.keys(patch.shifts).reduce((res, userId) => {
                        if (patch.shifts[userId] !== "Empty")
                            res[`${"shifts"}.${userId}`] = patch.shifts[userId];
                        return res;
                    }, {})
                },
                $unset: {
                    ...Object.keys(patch.shifts).reduce((res, userId) => {
                        if (patch.shifts[userId] === "Empty")
                            res[`${"shifts"}.${userId}`] = "";
                        return res;
                    }, {})
                }
            });
    };

    eventEmitter.emit("schedule-update", { team });
    eventEmitter.emit("client-update", { team });
}
```

Рисунок 2.9 – Комплексний обробник зміни розкладу

2.6 Механізми безпеки

Розглянуті вище проміжні обробники виконують важлиу роль під час встановлення того, чи дозволено тому чи іншому користувачу виконувати надісланий запит. Існує чотири унікальних проміжних обробника, що включають в себе обробники authenticateToken, authorizeUser, authorizeTeam та authorizeAdmin. Майже кожен запит (окрім тих, що доступні не автентифікованим клієнтам) починається з перевірки токену за допомогою проміжного обробника authenticateToken (рисунок 2.10).

```
5   function authenticateToken(req, res, next) {
6       const authHeader = req.headers['authorization'];
7       const token = authHeader && authHeader.split(' ')[1];
8       if (token == null) {
9           return res.status(401).json({ message: "Unauthorized" });
10      }
11      jwt.verify(token, process.env.JWT_TOKEN_SECRET, (error, user) => {
12          if (error) {
13              return res.status(401).json({ message: "Unauthorized" });
14          }
15          req.user = user;
16          next();
17      });
18  }
```

Рисунок 2.10 – Проміжний обробник authenticateToken

Цей обробник перевіряє, чи був токен насправді підписаний сервером, та відхиляє запит за умови хибного результату перевірки.

Система використовує JWT токени, що дозволяють за необхідрості гнучко горизонтально масштабувати сервер, оскільки не прив’язані до жодної сесії. Вони несуть в собі всю необхідно інформацію, що дозволяє однозначно ідентифікувати сутність, яку вони представляють, та перевірити автентичність за допомогою підпису.

Але успішна валідація токену ще не означає, що користувач справді ще існує та має доступ до запиту, який виконує. Для цього існує решта обробників, які ствять на меті визначити, чи існує користувач, чи належить він до команди, інформацію про яку він намагається отримати чи змінити, та чи має достатньо прав для здійснення такої операції.

Звісно, перед використанням токену, його необхідно згенерувати. Цим займаються модулі Signup, Signin. Перший відповідає за реєстрацію користувача, кодування одношляховим алгоритмом та збереження його паролю. Другий – за перевірку правильності ввденеих даних та генерацію

токену за умови, що логін та пароль користувача співпадають зі збереженими в базі значеннями. Додаткову функцію виконує модуль Signup, що перед збереженням нового користувача до бази звертається до Slack API з метою отримати ідентифікатор користувача в системі Slack.

2.7 Двигуни та планування

Сервіс використовує два модулі-двигуни, що покликані планувати події, як то створення нової черги або розсилка повідомлення.

Двигун Schedule відповідає за вчасне оновлення поточного розкладу, та орієнтується на вихідні дні команди, щоб не створювати зайвих черг. Кожного разу після запуску сервера, двигун перевіряє базу даних та за необхідності оновлює розклад (рисунок 2.11). Окрім того, він рекурсивно планує наступні оновлення, що відбудуться одразу після закінчення поточного розкладу. Таким чином, для кожної команди завжди існує один timeout, що чекає на оновлення.

Окрім планування, двигун також відповідає за нормалізацію черги, що відбувається кожного разу при одній з наступних змін: реєстрація/видалення користувача, зміна статусу користувача в календарі в день, якого стосується розклад, переміщення користувача в блок No Duty (блок для користувачів, що присутні, але за певних обставин не беруть участь в чергуванні). Цей функціонал реалізовано за допомогою івентів, які викликаються кожного разу при виконанні однієї з наведених змін.

Якщо підсумувати, функція, що відповідає за нормалізацію черги виконується кожного разу, коли отримує певний івент. Такий івент, в свою

чергу, генерується кожного разу, коли відбувається зв’язана з ним подія, як, наприклад оновлення календаря в розділі 2.5.

```
async function planSchedule(team) {
    clearTimeout(timeouts[team._id]);

    if (!(await scheduleManager.getAll(team._id)).length) {
        const newScheduleStartTime = new Date();
        newScheduleStartTime.setUTCHours(team.scheduleStartTime);
        newScheduleStartTime.setMinutes(0), newScheduleStartTime.setSeconds(0), newScheduleStartTime.setMilliseconds(0);
        await scheduleManager.create(
            team,
            getNextWorkingDay(newScheduleStartTime, team)
        );
    }
    const currentSchedule = await scheduleManager.getCurrent(team._id);
    const currentScheduleStartTime = new Date(currentSchedule.startTime);
    const currentScheduleEndTime = new Date(currentSchedule.startTime);
    currentScheduleEndTime.setHours(currentScheduleEndTime.getHours() + team.scheduleDuration);
    const countdown = currentScheduleEndTime - new Date();
    timeouts[team._id] = setTimeout(async () => {
        const newScheduleStartTime = new Date(currentScheduleStartTime);
        newScheduleStartTime.setDate(newScheduleStartTime.getDate() + 1);
        newScheduleStartTime.setUTCHours(team.scheduleStartTime);
        await scheduleManager.create(
            team,
            getNextWorkingDay(newScheduleStartTime, team)
        );
        const schedules = await scheduleManager.getAll(team._id);
        if (schedules.length > 3) {
            await scheduleManager.deteleOldest(team._id);
        }
        planSchedule(team);
    }, countdown);

    function getNextWorkingDay(date, team) {
        const interimDate = new Date();
        date.setUTCHours(team.scheduleStartTime + team.scheduleDuration);
        while (team.weekends.includes(date.toLocaleString('en-us', { weekday: 'long' })) || date - interimDate < 0) {
            date.setDate(date.getDate() + 1);
        }
        date.setUTCHours(team.scheduleStartTime);
        return date;
    }
}
```

Рисунок 2.11 – Функція planSchedule відповідає за вчасне оновлення розкладу для кожної з команд

Другий двигун – Notifications, працює за схожим принципом. Він аналізує чергу, вираховує час, в який необхідно відправити повідомлення, та звертається до Slack боту в момент, коли час настає. Поточні налаштування дозволяють користувачу отримати повідомлення за 5 хвилин до початк, на початку та одразу після закінчення чергування (рисунок 2.12).

Рисунок 2.12 – Приклад отриманих повідомлень від боту в Slack

2.8 Механізм оновлення інтерфейсу в режимі реального часу

Для досягнення ефекту оновлення в реальному часу використовуються Server-Sent Events. Клієнт підписується на отримання будь-яких повідомлень про зміни в його команді одразу після ініціалізації (рисунок 2.13). Кожного разу при отриманні повідомленні клієнт оновлює всі данні, що необхідні для відображення найновіших даних за допомогою функції populateData.

```
35    useEffect(() => {
36        if (!currentUser) {
37            return;
38        }
39        const eventSource = new EventSource(`/api/update/${currentUser.team}`);
40
41        eventSource.onmessage = (event) => {
42            const data = JSON.parse(event.data);
43            if (data.message = "update") {
44                populateData(false);
45            }
46        }
47
48        return () => {
49            eventSource.close();
50        }
51    }, [currentUser]);
```

Рисунок 2.13 – Підписка на отримання повідомлень на стороні клієнта

Сервер, в свою чергу, бере на себе відповідальність надсилати клієнту таке повідомлення кожного разу, коли відбуваються цікаві клієнту зміни (рисунок 2.14). Такі зміни включають в себе реєстрацію або видалення нового користувача та зміну в черзі або розкладі. Звісно, повідомлення отримують тільки учасники тієї команди, в якій відбулися зміни, оминаючи повідомлень про зміни в інших командах.

```
3   function update(req, res) {
4       res.set({
5           'Cache-Control': 'no-cache',
6           'Content-Type': 'text/event-stream',
7           'Connection': 'keep-alive',
8           'Access-Control-Allow-Origin': '*'
9       });
10      res.flushHeaders();
11
12      let lastSentTime = new Date();
13
14      function notify({ team }) {
15          if (team === req.params.team) {
16              if (new Date() - lastSentTime < 100) {
17                  return;
18              }
19              lastSentTime = new Date();
20              res.write(`data: { "message": "client-update" }\n\n`);
21          }
22      }
23
24      eventEmitter.on("client-update", notify);
25
26      res.on('close', () => {
27          eventEmitter.off("client-update", notify);
28          res.end();
29      });
30  }
```

Рисунок 2.14 – Функція, відповідальна за розсилку повідомлення всім підписаним користувачам, в чий команді відбулася зміна

2.9 Огляд клієнту

Клієнтська частина додатку повністю реалізована на React та містить в собі перелік вкладених компонентів (рисунок 2.14), необхідних для відображення всієї інформації.

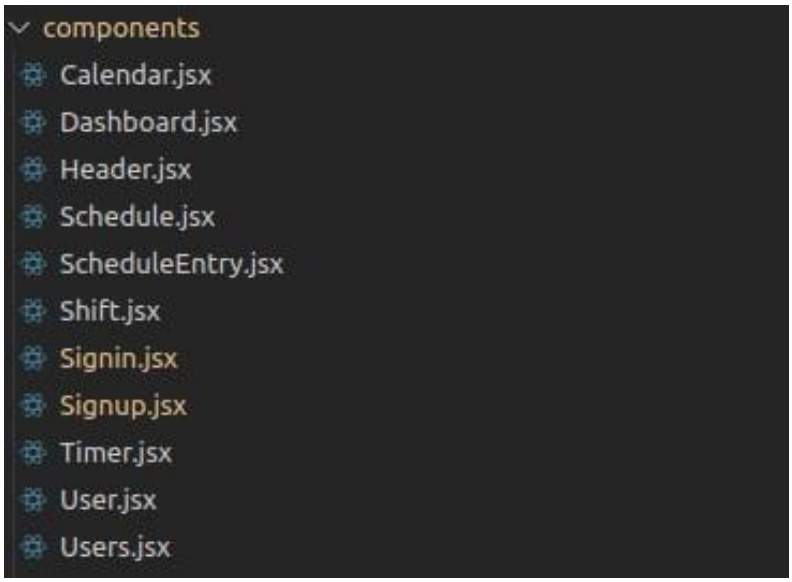

Рисунок 2.14 – React компоненти

Основними можна вважати компоненти Dashboard та Calendar, які відповідають за відображення черги та календарю відповідно. Вони ж займаються надсиланням та обробкою запитів та реагуванням на Server-Sent Events, згадані раніше.

Компоненти Signup та Signin покликані надати користувачу можливість зареєструватися та авторизуватися в системі, а всі інші слугують як вкладені компоненти ієрархії.

Навігація по сторінкам здійснюється за допомогою стандартної бібліотеки react-router-dom (рисунок 2.15).

```
 9  function App() {
10    return (
11      <Router>
12        <Routes>
13          <Route path="/" exact element={<Dashboard />} />
14          <Route path="/calendar" exact element={<Calendar />} />
15          <Route path="/signup" exact element={<Signup />} />
16          <Route path="/signin" exact element={<Signin />} />
17        </Routes>
18        <WavesBackground />
19      </Router>
20    );
21  }
```

Рисунок 2.15 – Шляхи до сторінок додатку

При зміні даних та відправленні запиту на сервер, клієнт використовує «оптимістичну стратегію», миттєво змінюючи стан даних на клієнті не чекаючи від сервера відповіді про їх успішну зміну. Таким чином користувач не бачить затримки між виконанням дії та отриманням результату. Втім, за допомогою повідомлень про оновлення, користувач майже одразу тримує актуальну версію даних вже з серверу.

2.10 Огляд інтерфейсу та функціоналу

На сторінці календаря користувач має змогу переглянути та змінити розклад, використовуючи прямокутне виділення (схоже до виділення на робочому столі Windows) та одну з дій, запропонованих в розділі Actions (рисунок 2.16).

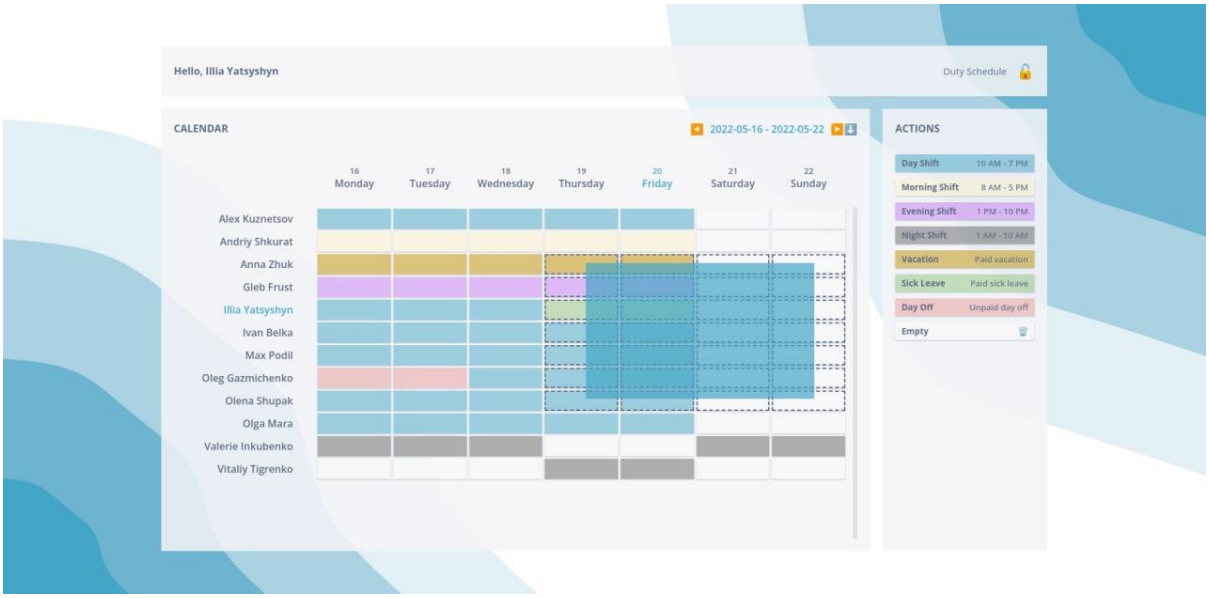

Рисунок 2.16 – Дія виділення на сторінці календаря

Сторінка з чергою, в свою чергу, надає можливість просто перетягнути одного з присутніх користувачів до бажаного інтервалу в розкладі, або прибрати його в блок No Duty таким же чином (рисунок 2.17).

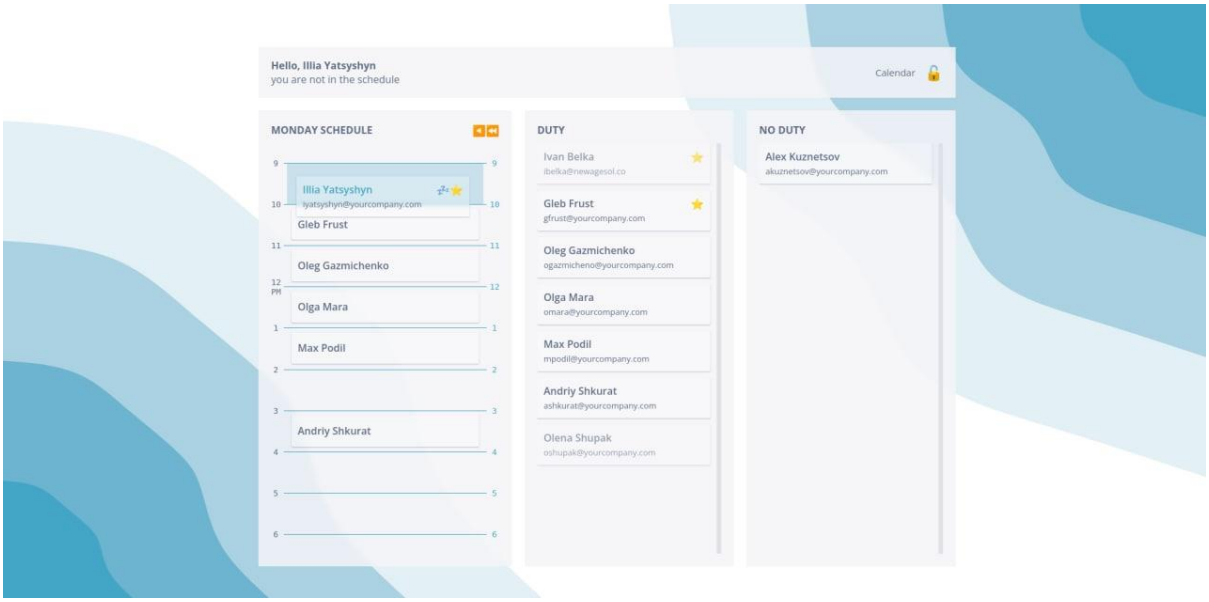

Рисунок 2.17 – Дія переміщення на сторінці розкладу

Обидві сторінки надають секцію для навігації та можливості вийти зі свого кабінету.

Реєстрація та авторизація зображені на рисунках 2.18 та 2.19 відповідно.

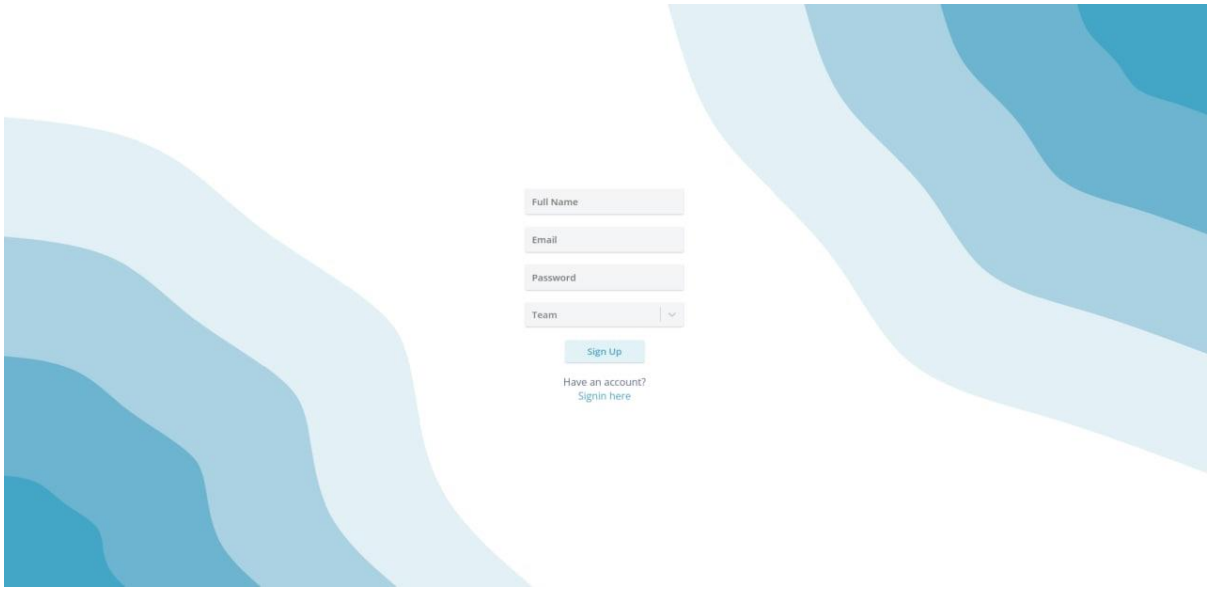

Рисунок 2.18 – Сторінка реєстрації

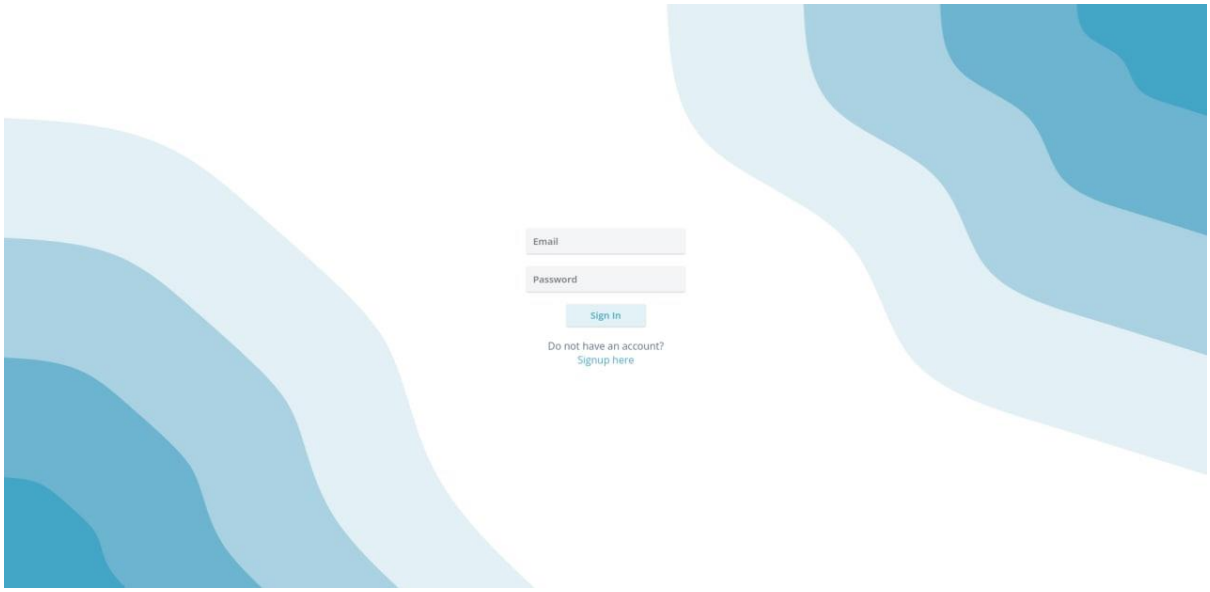

Рисунок 2.19 – Сторінка авторизації

Висновки

Під час виконання курсової роботи було створено зручну та досить потужну систему, що може стати в нагоді різним командам в кардинально відмінних сферах праці. Система захищена від несанкціонованого доступу та використовує сучасні методи захисту.

Для написання було використано сучасні технології та методи, що дозволили зробити розробку швидкою та комфортною.