КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра інтелектуальних програмних систем

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 121 «Інженерія програмного забезпечення»

на тему:

РОЗРОБКА ВЕБ ТА МОБІЛЬНОГО ЗАСТОСУНКУ ДЛЯ ВІДСТЕЖЕННЯ ЧАСУ

Виконав студент 4-го курсу

Антон ГЛАДКИЙ

(підпис)

Науковий керівник:

асистент, кандидат фіз.-мат. наук

Костянтин ЖЕРЕБ

(підпис)-

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем

«25» травня 2022 р.,

протокол № 10

Завідувач кафедри

Олександр ПРОВОТАР

(підпис)

Київ - 2022

РЕФЕРАТ

Обсяг роботи 41 сторінка,31 ілюстрація ,31 джерел посилань

Ключові слова: КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА,ВЕБ-ЗВСТОСУНОК,МОБІЛЬНИЙ-ЗАСТОСУНОК, РОБОЧІ ГОДИНИ, MVC, REACT,REACT-NATIVE, ПАНЕЛЬ АДМІНІСТРАТОРА, HEROKU

Об’єктом розробки є клієнт-серверний та мобільний застосунок для зручного вимірювання часу роботи.

Мета роботи полягає у розробці клієнт-серверного та мобільного застосунку, який забезпечить можливість розробнику надавати інформацію про свої робочі години та проект менеджеру аналізувати цю інформацію.

Методи та інструменти розробки. Для розробки сервера було використано Ruby з Ruby on Rails Framework. Для розробки клієнту було використано React. Для розробки мобільної додатку було використано React-Native. Для розгортання серверу використаний провайдер хмарних послуг Heroku.

Новизна роботи полягає у найбільш зручному для користувача інтерфейсі та панелі суперадміністратора для аналізу всіх даних. Та можливості використання результату розробки як повноцінного продукту.

В результаті було спроектовано та розроблено клієнт-серверний та мобільний застосунок з можливістю всіх дій які описані в меті роботи.

Практичне значення одержаних результатів. Застосунок такого типу може значно покращити результативність працівників в деякій компанії, допомогти проаналізувати в чому саме недоліки того чи іншого розробника, допомогти їх покращити.

ЗМІСТ

ВСТУП 4

1. ОПИС ВИКОРИСТАНИХ ТЕХНОЛОГІЙ 5

1.1 Вибір мови програмування для створення сервера 5

1.2 Ruby on Rails Framework 6

1.3 React 8

1.4 React-Native 10

2. ПРОЕКТУВАННЯ ТА РОЗРОБКА КЛІЄНТ-СЕРВЕРНОГО ЗАСТОСУНКУ 13

2.1 Розробка структури бази даних 13

2.2 Сервер 14

2.3 Клієнт 18

2.4 Мобільний додаток 20

2.5 Додаткова функціональність 27

3. ПОРІВНЯННЯ З ІНШИМИ СХОЖИМИ ЗАСТОСУНКАМИ 29

3.1 Порівняння з додатком Work Log Tracker 29

3.2 Порівняння з додатком Simple Time Tracker 31

3.3 Порівняння з додатком Jira 34

4. ДІЇ НАД ПРОЕКТОМ ПІСЛЯ РОЗРОБКИ 36

4.1 Тести та continuous integration 36

4.2 Розгортання на сервері Heroku 37

ВИСНОВОК 38

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 39

ВСТУП

Часто в багатьох компаніях особливо в сфері розробки програмного забезпечення, керівникам потрібно знати чи ефективно працівники витрачають свій час, на що саме, щоб потім знати що можна покращити або зі сторони компанії або зі сторони працівника.

Саме для цього і був створений мій додаток. Який дозволяє працівникам заповняти дані про розподіл свого часу в робочі години.

Звісно найбільш простим для використання буде веб-додаток тому для реалізації було обрано структура клієнт-серверного додатку.

Актуальність роботи полягає у важливості інформації про ефективність роботи працівників в компанії та інформації про те що саме потрібно покращити для певного працівника, тобто інформація про його недоліки.

Мета й завдання роботи. Метою роботи є розробка клієнт-серверного та мобільного застосунку та реалізації в ньому основних засобів збереження інформації, її обробки та передачі. Для досягнення мети були поставлені та виконані наступні **завдання**:

1. створення клієнт серверного застосунку;
2. створення мобільного застосунку
3. моделювання та розробка бази даних;
4. впровадження та реалізація найбільш ефективних способів передачі даних між клієнтом та сервером;
5. розробка панелі адміністратора для швидкого доступу до всіх даних та їх модифікації;
6. розгортання додатку на сервері Heroku для вільного доступу до нього;
7. додавання continuous integration для забезпечення постійної перевірки коректності програми.

Об’єктом є процес відслідковування робочого часу.

Предметом є застосунок для відслідковування часу.

1. ОПИС ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

1.1 Вибір мови програмування для створення сервера

Зазвичай клієнт-серверні застосунки – це досить великі за обсягом програмні рішення, які з часом масштабуються, доповнюються функціоналом. Серед варіантів мов для написання серверів для веб додатків слід відзначити Python, Java та Ruby.

За статистикою зібраною за допомогою Github (рис1.1). Java досить популярна та продуктивна мова програмування для побудови клієнт-серверних додатків. Причиною обрання не її як мовою серверної частини була наявність більшого досвіду саме з Ruby on Rails framework порівнянно з Spring framework. Python також досить популярний і має велику за обсягом користувацьку базу але не був обраний по причині відсутності у мене відмінних від базових знань та практики в цій мові програмування.

# Ranking	Programming Language	Percentage (YoY Change)	YoY Trend
1	JavaScript	18.756% (+0.053%)	
2	Python	16.628% (+0.390%)	
3	Java	11.680% (+0.742%)	
4	Go	7.829% (-1.176%)	
5	Ruby	7.588% (+0.776%)	^
6	C++	6.985% (-0.439%)	v
7	TypeScript	6.604% (-0.164%)	
8	PHP	5.081% (-0.046%)	
9	C#	3.614% (-0.221%)	
10	C	3.253% (+0.072%)	
11	Shell	2.191% (+0.380%)	^

Рисунок 1.1 − Популярність мов відповідно до кількості pull requests на ресурсі GitHub за 1 квартал 2021 року [1]

Однією з найбільш популярних мов є саме Ruby. Це повністю об’єктно-орієнтовна мова програмування з динамічною типізацією. На відміну від Java змінні в Ruby не мають чітко заданого типу, що на мою думку полегшує роботу. Але також в багатьох речах мова програмування дуже схожа на Java, що

полегшило розробку застосунку цією мовою. І звісно основним критерієм вибору саме Ruby став фреймворк Ruby on Rails який орієнтований для написання клієнт-серверних додатків.

1.2 Ruby on Rails Framework

Для розробки веб додатків написаних на мові Ruby будемо використовувати Ruby on Rails Framework [2]. Цей фреймворк об’єднує в собі весь потрібний функціонал для розробки клієнт-серверних додатків. Основними перевагами цього фреймворку при розробці клієнт-серверних систем є:

1. Модель-Вигляд-Контролер (Model-View-Controller) [3]: програмний шаблон, а в даному випадку каркас на основі HTTP сервлета, що забезпечує можливості для створення RESTful веб-додатків і веб-служб.
2. Управління та доступ до даних – можливість працювати з SQL та NoSQL базами даних.
3. Можливості для написання юніт тестів та інтеграційних тестів.

Переваги Ruby on Rails на цьому не закінчуються. Найважливішою частиною цього фрейморку, яка буде в даній роботі і використовується у кожному веб-додатку є шаблон MVC. Він дозволяє нам створити на сервері певну модель даних та реалізувати RESTful сервіс для того щоб ми могли звертатись до даних, обробляти ці дані, додавати нові дані, зберігати дані до бази даних тощо за допомогою HTTP запитів з клієнту. Для цього на сервері нам потрібно буде створити набір контролерів, що будуть приймати й обробляти запити, посилати відповіді(рис 1.2).

```ruby
class PeopleController < ActionController::Base
  # This will raise an ActiveModel::ForbiddenAttributesError exception
  # because it's using mass assignment without an explicit permit
  # step.
  def create
    Person.create(params[:person])
  end

  # This will pass with flying colors as long as there's a person key
  # in the parameters, otherwise it'll raise an
  # ActionController::ParameterMissing exception, which will get
  # caught by ActionController::Base and turned into a 400 Bad
  # Request error.
  def update
    person = current_account.people.find(params[:id])
    person.update!(person_params)
    redirect_to person
  end

  private
    # Using a private method to encapsulate the permissible parameters
    # is just a good pattern since you'll be able to reuse the same
    # permit list between create and update. Also, you can specialize
    # this method with per-user checking of permissible attributes.
    def person_params
      params.require(:person).permit(:name, :age)
    end
end
```

Copy

Рисунок 1.2 − Приклад контролера написаного, використовуючи Ruby on Rails [4]

Для забезпечення зв'язку між деяким запитом та методом в контролері використовується конфігураційний файл routes.rb. Додавши **resources: photos** отримаємо такі можливі запити з відповідними методамиконтролера(рис 1.3).

HTTP Verb	Path	Controller#Action	Used for
GET	/photos	photos#index	display a list of all photos
GET	/photos/new	photos#new	return an HTML form for creating a new photo
POST	/photos	photos#create	create a new photo
GET	/photos/:id	photos#show	display a specific photo
GET	/photos/:id/edit	photos#edit	return an HTML form for editing a photo
PATCH/PUT	/photos/:id	photos#update	update a specific photo
DELETE	/photos/:id	photos#destroy	delete a specific photo

Рисунок 1.3 − Приклад запитів отриманих за допомогою файлу routes.rb [5]

1.3 React

Окрім сервера нам потрібно реалізувати і клієнт. Сьогодні популярним для веб-сайтів є SPA (Single-page application) [6]. Основна відмінність та перевага односторінкових додатків є те, що весь код розмітка (HTML), стилі (CSS) та функції (Scripts) завантажуються лише один раз під час запуску додатку, і надалі мережу будуть навантажувати тільки дані отримані чи відправленні за допомогою HTTP запитів. Це покращує швидкодію системи, та усуває постійне завантаження сторінки під час переходу на нову неї, що в свою чергу покращує досвід користувача.

Зважаючи на вище сказане, будемо обирати бібліотеку JavaScript для побудови SPA веб-сайтів. Звернувшись до статистики популярності (рис. 1.3), бачимо що на сьогодні React [7] є найпопулярнішою бібілотекою JavaScript для створення односторінкових додатків і його популярність зростає з кожним роком досить стрімко.

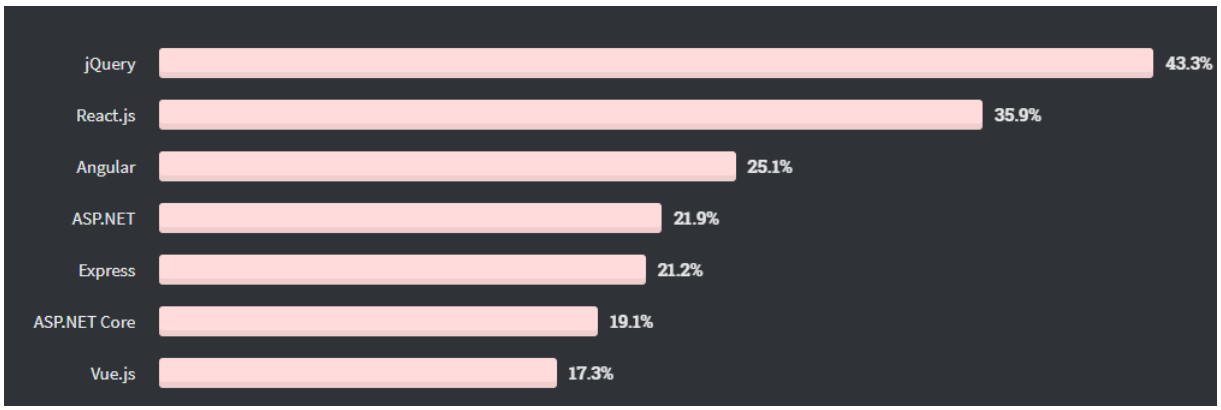

Рисунок 1.3 − Популярність front-end бібліотек згідно опитування на ресурсі Stack Overflow [8]

Серед переваг React слід відмітити:

1) Декларативність. React самостійно оновлює відповідні дані згідно до того як це задав розробник.

2) Компоненти – React дозволяє створювати інкаспульовані компоненти, які потім легко об’єднуються в більш складні сторінки, передавати в них різні аргументи.

3) Велика кількість бібліотек

4) Відкритий код

Розглянемо альтернативу React: Angular [9].

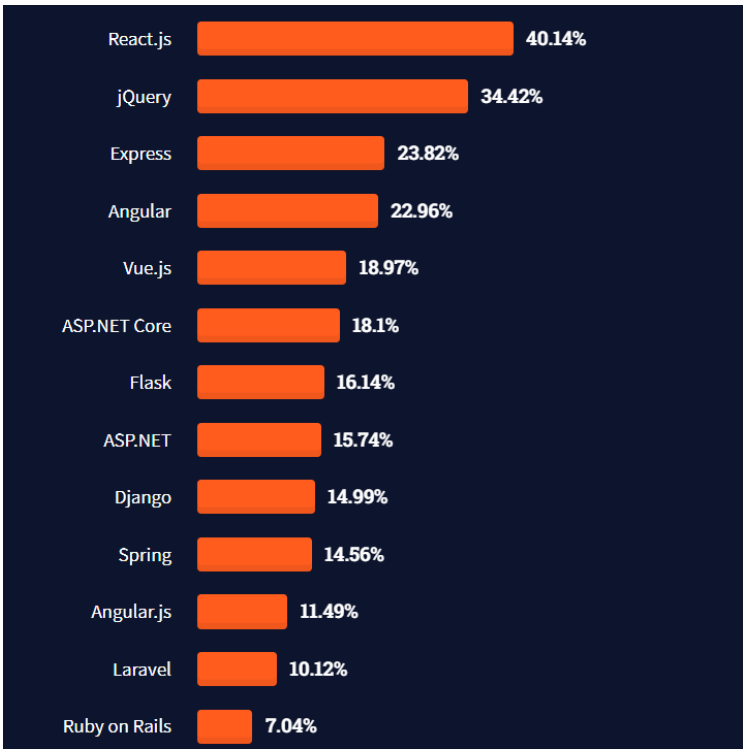

Рисунок 1.4 − Популярність web frameworks згідно опитування на ресурсі Stack Overflow [10]

Порівнюючи популярність(рис. 1.3) очевидно React використовується частіше, що в свою чергу є плюсом.

Переваги React відносно Angular:

1) Продуктивність через використання virtual DOM [11]
2) Менший розмір скомпільованого проекту
3) Можливість легко переключатися між версіями

Недоліки React відносно Angular:

1) Більш складно інтегрувати з MVC framework

1.4 React-Native

Загалом клієнтської частини у вигляді веб-застосунку достатньо для зручного користування програмою, але для максимальної зручності та доступу до застосунку в моменти знаходження не біля комп'ютера було вирішено розробити мобільний застосунок який буде виконувати однакову функцію і оперувати даними якими оперує веб-застосунок.

Для розробки даного застосунку була в обраний framework React-Native , звісно це не єдиний варіант для розробки подібного типу програм тому давайте порівняємо даний framework з альтернативами.

Альтернативним фреймворком є Flutter [12]. По перше порівнюючи популярність двох фремворків (рис. 1.5) можна зазначити, що вони мають однакову популярність серед розробників.

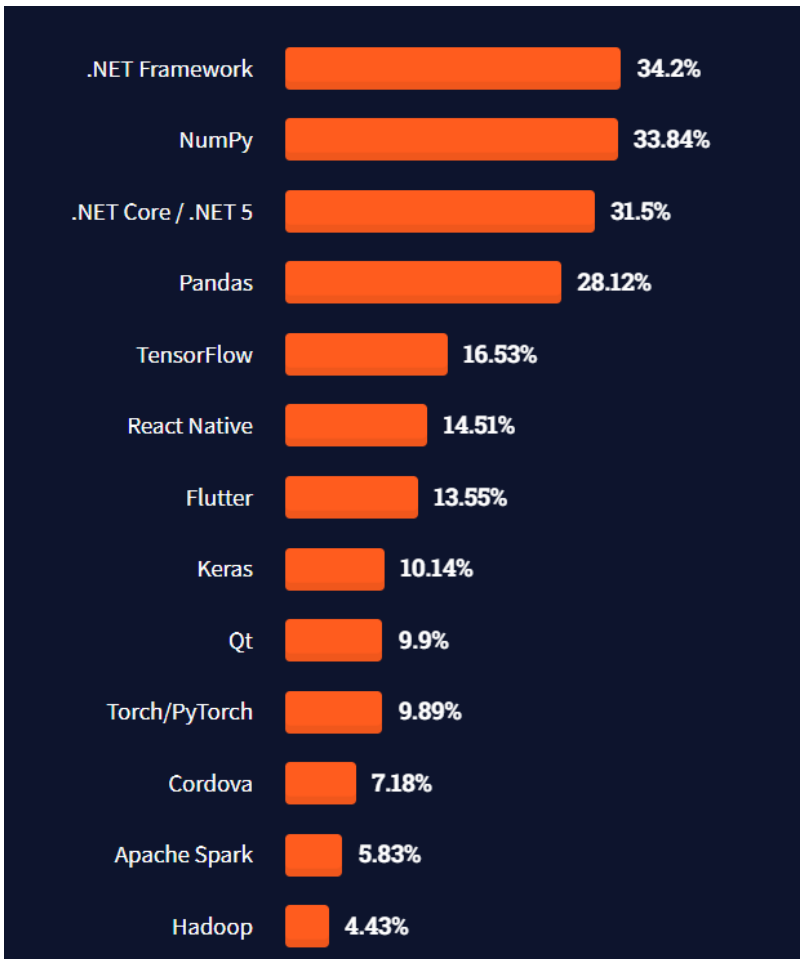

Рисунок 1.5 – Популярність фреймворків згідно опитування на ресурсі Stack Overflow [13]

Порівнюючи інші показники такі як продуктивність Flutter має кращі показники через те що код повністю компілюється в нативний код порівнюючи з React-Native де частина коду залишається в вигляді JavaScript .

Також загалом Flatter має більш складну структуру що для мене стало мінусом під час вибору фреймворку.

Інші фремворки менш популярні порівняно з двома названими попередньо, тому їх я не розглядав як можливі варіанти.

Ще однією альтернативою було написання коду відразу на нативній мові, в випадку Android це була б Java.

В цьому випадку основною проблемою є те що React-Native дає змогу розробляти застосунок відразу для систем Android та IOS пишучи код один раз, що не можливо в випадку розробки нативною мовою, також через те що React-

Native відпрацьовує код JavaScript в окремому потоці це покращує взаємодію з інтерфейсом. [14] Отже навіть не дивлячись на перевагу нативної мови в продуктивності React-Native всерівно виступає як найкращий фреймворк для такого типу розробки.

Звісно ще одним основним критерієм виступає схожість React та React-native що спрощує розробку.

2. ПРОЕКТУВАННЯ ТА РОЗРОБКА КЛІЄНТ-СЕРВЕРНОГО ЗАСТОСУНКУ

2.1 Розробка структури бази даних

В даному застосунку я використав PostgreSQL в ролі бази даних для збереження такої інформації: як проекти, які зараз знаходяться в розробці, різні можливі категорії діяльності наприклад: розробка, тестування, перерва, виправлення багів тощо. Також зберігається інформація про користувачів у яких відповідно є дві доступні ролі: звичайний користувач(розробник) та проект менеджер(адмін). Основною табличкою все таки є табличка де зберігається який розробник коли, скільки часу та що робив. І звісно також маємо інформацію про користувачів у яких більше доступу ніж у звичайного адміна, назвемо їх суперадміном. Саме ці користувачі мають вільний доступ до всіх даних та їх зміні.

Розглянемо таблички в базі даних:

Основні поля таблиць CATEGORIES та PROJECTS це поле name, яке відповідає імені відповідного проекту або категорії(типу діяльності)

USERS - табличка де зберігається інформація про розробників та звичайних адмінів (проект менеджерів)(рис. 2.1):

Columns

	Name	Data type
	id	bigint
	email	character varying
	encrypted_password	character varying
	created_at	timestamp without time zone
	updated_at	timestamp without time zone
	firstName	character varying
	lastName	character varying
	role	integer

Рисунок 2.1 − Поля таблички Users

Основними полями є:

1) email - за допомогою нього відбувається логін 2) encrypted_password – зашифрований пароль 3) firstName,lastName – ім’я та прізвище користувача 4) role – роль (може бути або developer або admin).

ACTIVITIES(табличка де зберігається інформація про діяльність розробників)(рис. 2.2):

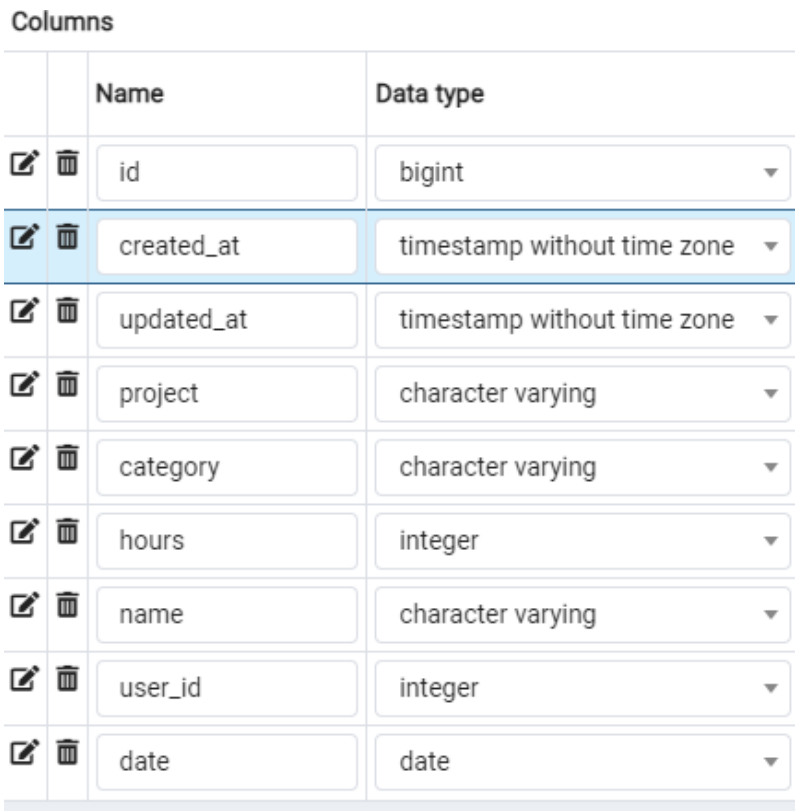

Рисунок 2.2 − Поля таблички Activities

Основними полями є:

1) project – проект де працював розробник під час цієї активності
2) category– що саме робив розробник
3) hours – кількість годин
4) user_id – id розробника який виконував цю дію.

2.2 Сервер

Після вибору технологій та моделювання бази даних переходимо до розробки серверної частини застосунку.

Для початку потрібно організувати аутентифікацію користувачів. Для цього було використана популярна бібліотека (гем) Devise[15]. В основному він використовується для застосунків написаних повністю на Ruby on Rails (далі скорочено RoR), тобто для таких застосунків, де клієнтська частина складається з багатьох html сторінок. Але так як я вирішив виконувати клієнтську частину на React, стандартно створена модель за допомогою Devise потребувала доопрацювання, а саме я обрав варіант аутентифікації за допомогою jwt токена[16].

Переглянемо код класу який відповідає за обробку токена(рис. 2.3):

```ruby
class JsonWebToken
  def self.encode(payload)
    expiration = 2.weeks.from_now.to_i
    JWT.encode payload.merge(exp: expiration), ENV['secretkey']
  end

  def self.decode(token)
    JWT.decode(token, ENV['secretkey']).first
  end
end
```

Рисунок 2.3 − Клас для роботи з JWT(JsonWebToken)

І також потрібно було перевизначити як саме відбувається аутентифікація(рис. 2.4):

```ruby
module Devise
  module Strategies
    class JWT < Base
      def valid?
        request.headers['Authorization'].present?
      end

      def authenticate!
        token = request.headers.fetch('Authorization', '').split(' ').last
        payload = JsonWebToken.decode(token)
        success! User.find(payload['sub'])
      rescue ::JWT::ExpiredSignature
        fail! 'Auth token has expired'
      rescue ::JWT::DecodeError
        fail! 'Auth token is invalid'
      end
    end
  end
end
```

Рисунок 2.4 − Перевизначення стандартної аутентифікації Devise

Після таких дій під можемо написати контролер який буде відповідати за аутентифікацію користувача приймаючи email та пароль через HTTP POST request. Розглянемо саме метод який відповідає за створення користувача (рис. 2.5):

```ruby
class AuthenticationController < ApiController
  skip_before_action :authenticate_user!, only: [:create]
  def create
    user = User.find_by(email: params[:email])
    if user&.valid_password?(params[:password])
      render json: { token: JsonWebToken.encode(sub: user.id),user: user}
    else
      render json: { errors: 'invalid' },status:401
    end
  end
end
```

Рисунок 2.5 − Метод аутентифікації після прийому запиту з email та паролем

В застосунку відсутній контролер та логіка для реєстрації нового

користувача. Це було зроблено спеціально для того щоб система булазакрита для посторонніх людей і нових юзерів міг додавати тільки суперадмін.

Для моделей Categories, Users, Activities присутній тільки запит на отримання всіх даних тому що загалом інших дій з ними не виконується звичайними користувачами. Але контролер моделі Activities більше складний розглянемо його детальніше(рис. 2.6):

```
def index
  if current_user.admin?
    @activity = Activity.where(:date => params[:date])
    if(params[:user_id]!='')
      @activity=@activity.where(:user_id=>params[:user_id])
    end
    if(params[:project]!='')
      @activity=@activity.where(:project=>params[:project])
    end
    if(params[:category]!='')
      @activity=@activity.where(:category=>params[:category])
    end

  elsif current_user.developer?
    @activity = current_user.activities.where(:date => params[:date])
  end
  @activity=@activity.offset((params[:page].to_i-1)*10).limit(10)
  render json: @activity
end
```

Рисунок 2.6 − Index method for Activities controller

Як буде далі пояснено в клієнтській частині присутні поля за якими можна фільтрувати activities, а вид подачі даних використовує пагінацію[17], тобто подання інформації сторінками, чому вибраний саме цей варіант буде пояснено далі. В методі бачимо що дані фільтрів та номер сторінки передаються через параметри запиту.

Інші методи контролеру відповідають стандартним CRUD операціям щодо моделі Activities.

2.3 Клієнт

Для клієнтської частини були поставлені такі задачі:

1. Відправляти запит на отримання всіх даних залежно від ролі користувача
2. Відправляти дані для аутентифікації використовуючи форму
3. Надання інтерфейсу для редагування, створення та видалення нових активностей для розробника та перегляд і фільтрування дляадміна
4. Відображення спінера коли йде очікування відповіді від серверу Розглянемо вигляд інтерфейсу для розробника(developer):

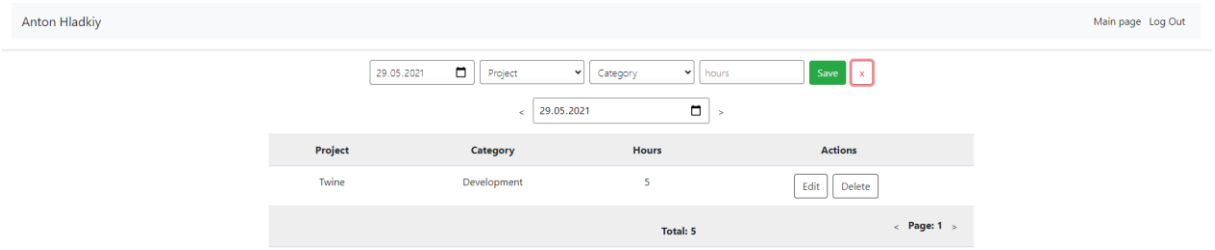

Рисунок 2.7 − Основний інтерфейс користувача з роллю developer

В інтерфейсі(рис. 2.7) спостерігаються такі компоненти:

1. Ім’я та прізвище користувача
2. Кнопка Log Out
3. Поля для створення нової активності
4. Кнопка збереження нової активності та кнопка очищення полів
5. Поле для вибору дат(воно є фільтром за допомогою якого користувач може переглянути свої минулі активності)
6. Таблиця з активностями за день вказаний вище
7. Кнопки редагування та видалення для активностей
8. Номер сторінки та кнопки для її перемикання Розглянемо вигляд інтерфейсу для адміна(admin):

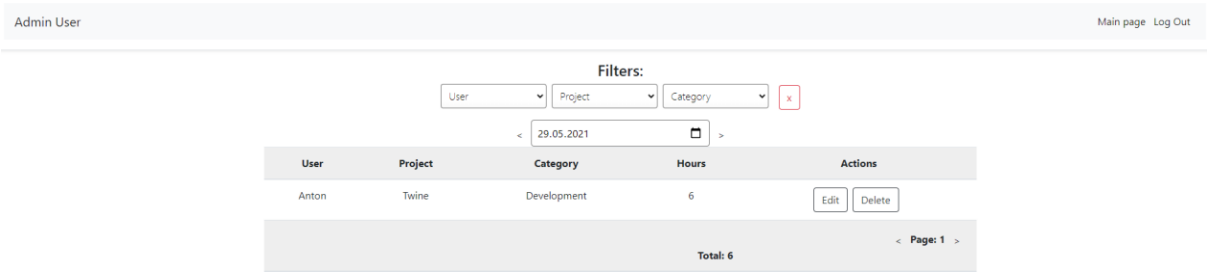

Рисунок 2.8 − Основний інтерфейс користувача з роллю admin

В інтерфейсі(рис. 2.8) спостерігаються такі компоненти:

1. Ім’я та прізвище користувача
2. Кнопка Log Out
3. Поля фільтрів які фільтрують всі дані по відповідним критеріям
4. Кнопка очищення фільтрів
5. Поле для вибору дат (воно є фільтром за допомогою якогокористувач може переглянути минулі активності)
6. Таблиця з активностями за день вказаний вище
7. Кнопки редагування та видалення для активностей
8. Номер сторінки та кнопки для її перемикання

Раніше згадувалося що було обрано спосіб передачі даних на клієнтську частину під назвою пагінація. Основна мотивація використати саме цей спосіб в тому що на стороні клієнту зберігається не вся інформація але лише її мала частина яка потрібна для активної сторінки.

Для аутентифікації створена проста форма яка містить два поля для вводу:

1. email
2. password

Відповідно ці дані відправляються на сервер, де обробляються і в результаті клієнтська частина отримує токен та роль користувача. Відповідно до ролі відображається або інтерфейс адміна або

розробника, а токен надається як header в всі наступні запити, для тогощоб вони виконувалися а не блокувалися на стороні серверу.

Також для перевірки правильності рендерингу компонентів були створені тести на основі утиліти enzyme [18]. Ця утиліта дає змогу відображати компоненти незалежно від інших та перевіряти що результат не змінився після

розробки інших компонентів.

Розглянемо приклад такого тесту(рис. 2.9):

```
import React from 'react';
import { shallow} from 'enzyme';
import Login from '../components/Login';
let initialUser={
    email:'',
    password:''
}
describe('Login', () => {
    let wrapped = shallow(<Login initialUser={initialUser} />);
    it('should render the Login Component correctly', () => {
        expect(wrapped).toMatchSnapshot();
    });
});
describe('Login when already loggenIn', () => {
    let wrapped = shallow(<Login initialUser={initialUser} loggedIn={true}/>);
    it('should render the Login Component correctly', () => {
        expect(wrapped).toMatchSnapshot();
    });
});
```

Рисунок 2.9 – Приклад тесту для клієнтської частини написаний за допомогою enzyme

2.4 Мобільний додаток

Для мобільного додатку були поставлені такі задачі:

1. Відправляти запит на отримання всіх даних залежно від ролі користувача
2. Відправляти дані для аутентифікації використовуючи форму
3. Надання інтерфейсу для редагування, створення та видалення нових активностей для розробника та перегляд і фільтрування дляадміна

Розглянемо вигляд інтерфейсу(рис. 2.10) для розробника(developer):

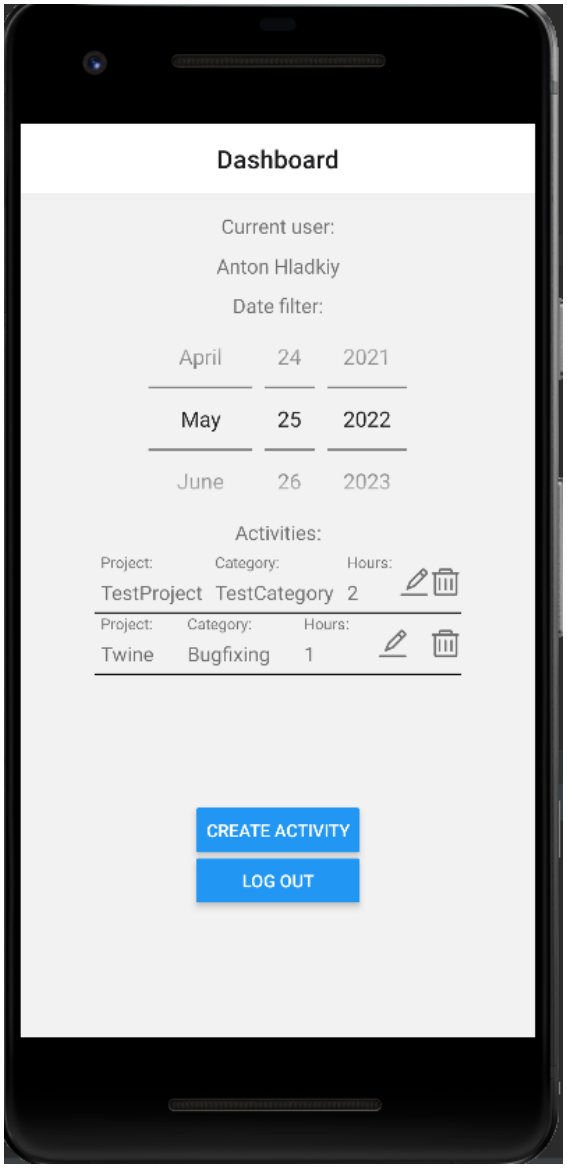

Рисунок 2.10 – Основний мобільний інтерфейс користувача з роллю developer

В інтерфейсі спостерігаються такі компоненти:

1. Ім’я та прізвище користувача
2. Кнопка Log Out
3. Кнопка для переходу в інтерфейс створення нової активності
4. Поле для вибору дати(воно є фільтром за допомогою якого користувач може переглянути свої минулі активності)
5. Список з активностями за день вказаний вище
6. Кнопки редагування та видалення для активностей

Розглянемо вигляд інтерфейсу(рис. 2.11) для адміна(admin):

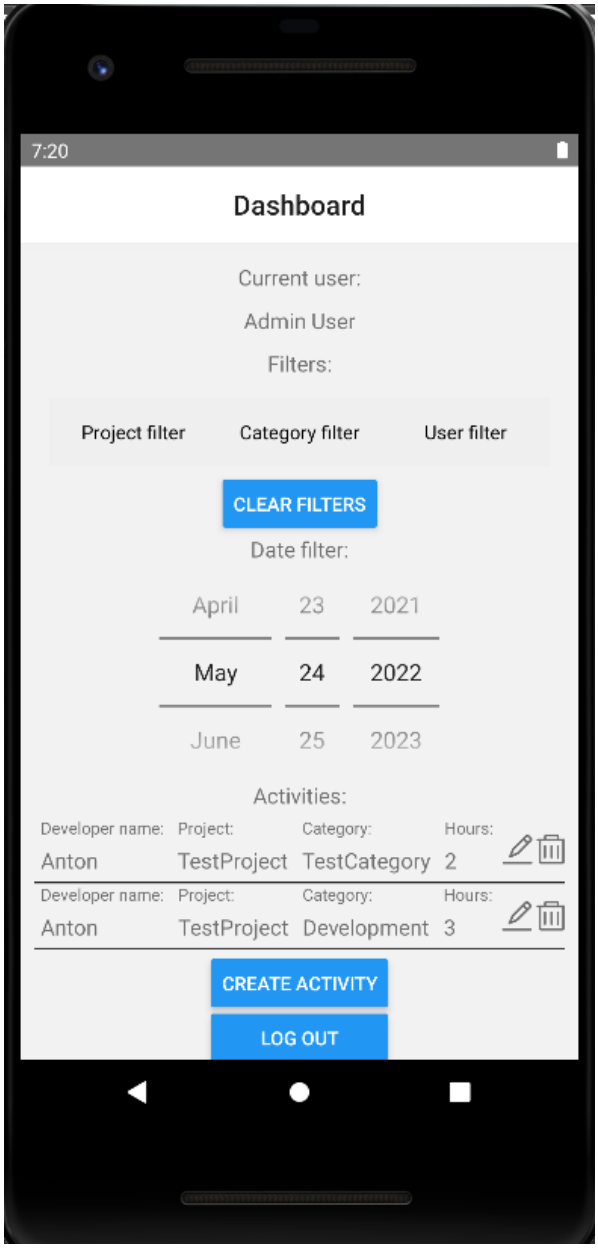

Рисунок 2.11 − Основний мобільний інтерфейс користувача з роллю admin

В інтерфейсі спостерігаються такі компоненти:

1. Ім’я та прізвище користувача
2. Кнопка Log Out
3. Поля фільтрів які фільтрують всі дані по відповідним критеріям
4. Кнопка очищення фільтрів
5. Поле для вибору дати (воно є фільтром за допомогою якого

користувач може переглянути минулі активності)

6. Таблиця з активностями за день вказаний вище
7. Кнопки редагування та видалення для активностей

Розглянемо вигляд інтерфейсу(рис. 2.12) для створення та редагування активностей:

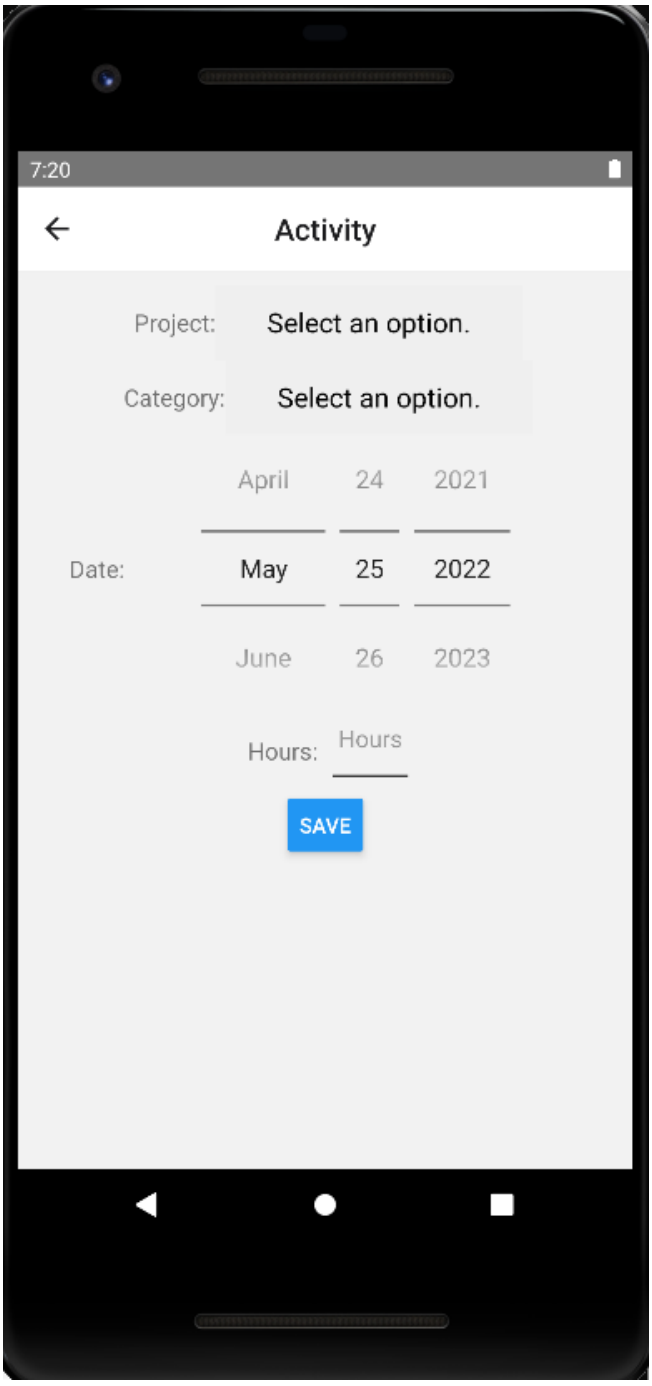

Рисунок 2.12 − Інтерфейсу для створення та редагування активностей

В інтерфейсі спостерігаються такі компоненти:

1. Поля заповнення даних
2. Кнопка для відправлення запиту

Для відображення списку активностей був використаний компонент FlatList [18], основна перевага якого в тому що виконується рендер тільки тих об’єктів які зараз відображаються що в свою чергу покращує продуктивність та покращує якість досвіду користування застосунком.

Для формування запитів було використано бібліотеку axios [19]. Також було написано деяке її покращення яке додати заголовок авторизації до кожного запиту автоматично.(рис. 2.13).

```
export const useCustomAxios = (): AxiosInstance => {
  const customAxios = axios.create();

  const getToken = async () => {
    try {
      const value = await AsyncStorage.getItem( key: 'token');
      if (value !== null) {
        return value;
      }
    } catch (e) {
      return undefined;
    }
  };
  customAxios.interceptors.request.use( onFulfilled: async (req: AxiosRequestConfig) => {
    await adminHeaderAdd(req);
    return req;
  });

  const adminHeaderAdd = async (req: AxiosRequestConfig) => {
    const acesToken: string | undefined = await getToken();
    console.log(acesToken);
    const headers = req.headers as {Authorization: string};
    headers.Authorization = acesToken ? acesToken : '';
  };
  return customAxios;
};
```

Рисунок 2.13 − Приклад застосування interceptor для бібліотеки axios

Деякі дані отримані в результаті запитів не потребують частого оновлення і також використовуються часто в програмі. Для даних такого типу було використано технологію hooks [20] а саме useContext [21] hook.

Ця технологія ідеально вирішує проблему описану вище, приклад застосування даного hook для збереження інформації про проекти (рис. 2.14).

```tsx
import React, {ReactElement, useEffect, useState} from 'react';
import {Project, useGetProjects} from '../api/projectApi';

interface ContextType {
  projects: Project[] | null;
  updateProjects: () => void;
}

export const ProjectContext = React.createContext({} as ContextType);

interface Props {
  children: React.ReactNode;
}

export const ProjectProvider = ({children}: Props): ReactElement => {
  const [projects, setProjects] = useState<Project[] | null>( initialState: null);
  const [updateProjects, projectsResponse] = useGetProjects();
  useEffect( effect: () => {
    if (projectsResponse) {
      setProjects(projectsResponse);
    }
  }, deps: [projectsResponse]);

  return (
    <ProjectContext.Provider
      value={{
        projects,
        updateProjects,
      }}>
      {children}
    </ProjectContext.Provider>
  );
};
```

Рисунок 2.14 – Приклад використання useContext

Також однією із проблем в розробці такого типу застосунків виступає збереження даних між сесіями користування додатку, саме в цьому випадку це дані потрібні для аутентифікації користувача (token). Без вирішення даної проблеми користувачу потрібно буде виконувати операцію логіну після кожного перезавантаження додатку. А для вирішення даної проблеми було використана бібліотека @react-native-async-storage/async-storage [22], однієї з функцій якої є саме вирішення даних проблем. Дана бібліотека зберігає дані в вигляді {key,value} де key та value мають тип даних string, тобто для зберігання об’єктів їх

спочатку потрібно перетворити на JSON string.

Приклад збереження та зчитування об’єкту(рис. 2.15):

```
const storeToken = async (token: string) => {
  try {
    await AsyncStorage.setItem( key: 'token', token);
  } catch (e) {}
};
const getToken = async () => {
  try {
    const value = await AsyncStorage.getItem( key: 'token');
    if (value !== null) {
      return value;
    }
  } catch (e) {
    return undefined;
  }
};
```

Рисунок 2.15 − Приклад застосування async-storage

Також корисною практикою є створення власних хуків(hooks) наприклад у моєму випадку хуки для відправлення запитів. Приклад такого коду для запиту GET(рис. 2.16):

```
import {useGetAll} from './template/useGetAll';

export interface Category {
  name: string;
}

export const useGetCategories = () => {
  return useGetAll<Category[]>( url: `${HOST}categories`);
};
```

```
const [updateCategories, categoriesResponse] = useGetCategories();
```

```
export const useGetAll = <RESPONSE_TYPE>(
  url: string,
  fetchAtRender : boolean  = true,
): [
  () => void,
  RESPONSE_TYPE | undefined,
  boolean | undefined,
  {message: string} | undefined,
] => {
  const [responseData, setResponseData] = useState<RESPONSE_TYPE>();
  const [isLoading, setIsLoading] = useState<boolean>();
  const [error, setError] = useState<{message: string}>();
  const [update, setUpdate] = useState<any>();
  const axios = useCustomAxios();

  useEffect( effect: () => {
    if (!url) return;
    if (!update && !fetchAtRender) return;
    (async () => {
      setResponseData( value: undefined);
      setError( value: undefined);
      setIsLoading( value: true);
      try {
        const resp = (await axios.get(url)).data;
        setResponseData(resp);
      } catch (e) {
        setError(e);
      }
      setIsLoading( value: false);
    })();
  }, deps: [url, update]);

  return [() => setUpdate( value: {}), responseData, isLoading, error];
};
```

Рисунок 2.16 − Example of custom hook

2.5 Додаткова функціональність

Для зручного створення нових юзерів та інших даних а також перегляду всієї інформації на сервері було створено інтерфейс для суперадміну. Створено за допомогою бібліотеки ActiveAdmin, яка досить проста в використанні і ефективна для відносно швидкого створення відповідної панелі. Але також створений таким чином інтерфейс дозволяє навіть завантажувати відповідні дані в різних форматах.

Дана функція доступна тільки для деякого одного користувача, наприклад це може бути директор компанії або проект менеджер.

Для доступу до цієї функції потрібно перейти за адресою /admin.

Загалом це можна зазначити як перевагу мого додатку порівняно з іншими варіантами існуючими на ринку.

3. ПОРІВНЯННЯ З ІНШИМИ СХОЖИМИ ЗАСТОСУНКАМИ

3.1 Порівняння з додатком Work Log Tracker

Розглянемо функціонал та інтерфейс Work Log Tracker [23] даного додатку. Спочатку звернемо увагу на інтерфейс (рис. 3.1, рис. 3.2, рис. 3.3, рис. 3.4):

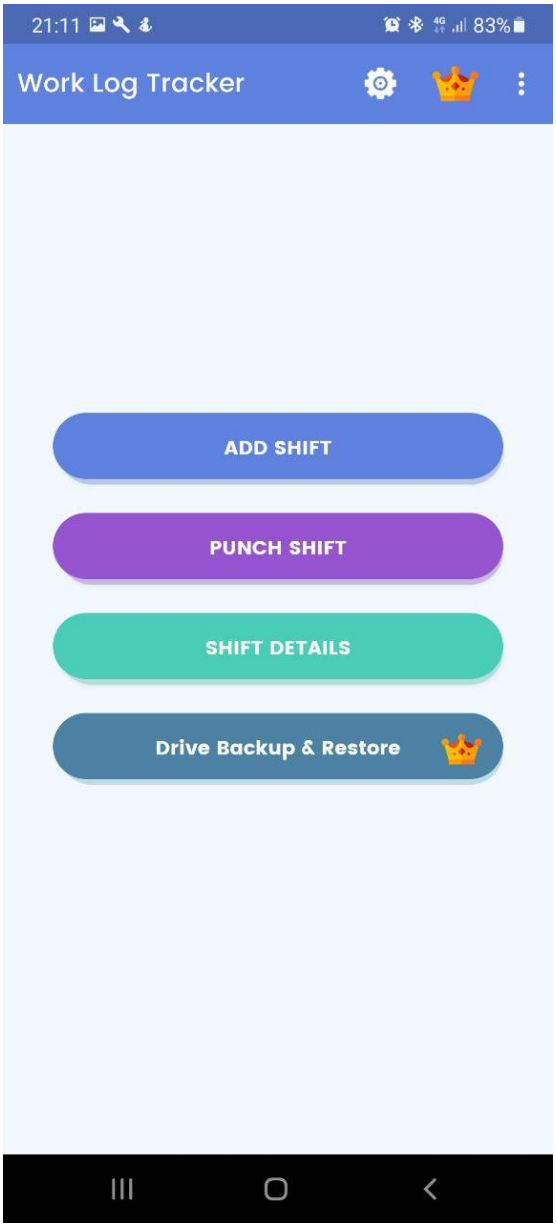

Рисунок 3.1 − Work Log Tracker dashboard

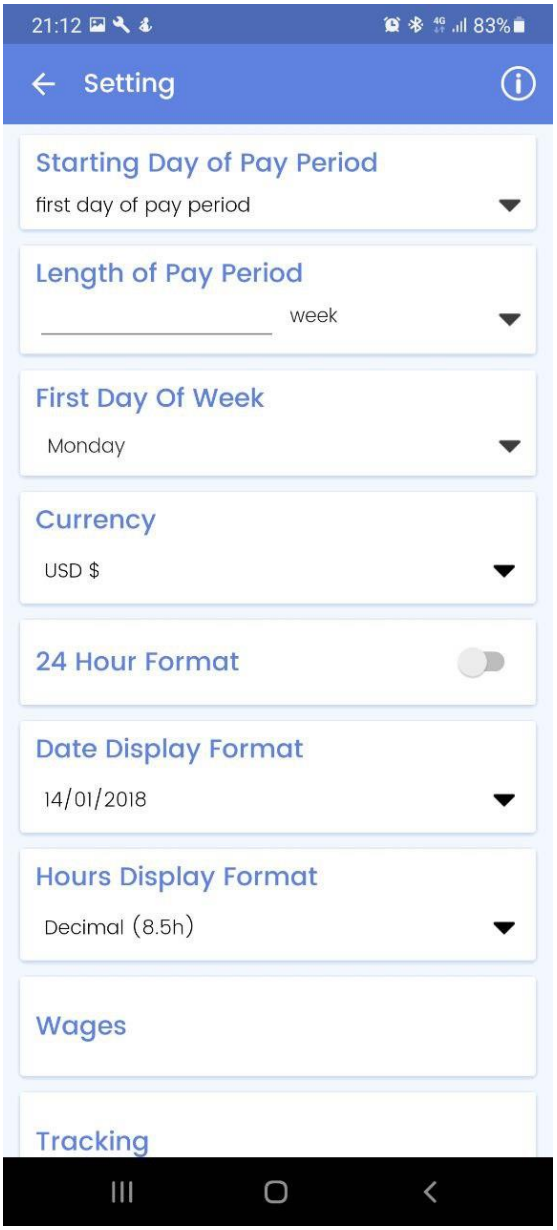

Рисунок 3.2 − Work Log Tracker settings

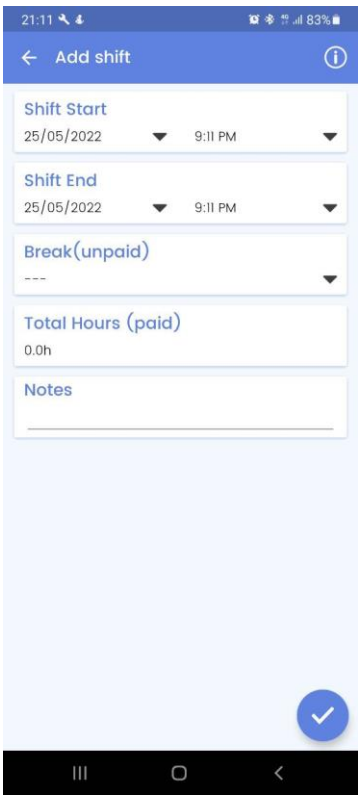

Рисунок 3.3 − Work Log Tracker add shift сторінка

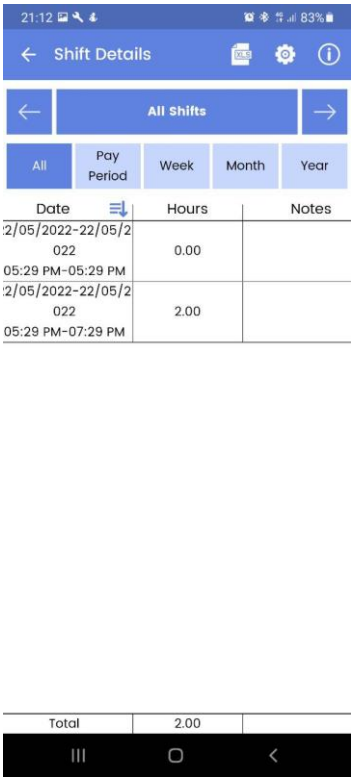

Рисунок 3.4 − Work Log Tracker shift details сторінка

Серед функціоналу можна помітити більшу кількість можливостей але в свою чергу це призводить до великої кількості сторінок, налаштувань та даних які

в свою чергу можуть бути складними для розуміння звичайному користувачеві.

Тому з переваг свого застосунку можна виділити такі пункти:

1. Зрозумілий та спрощений інтерфейс
2. Можливість використання веб застосунку в разі необхідності
3. Існування спеціальної ролі адміну який може переглядати дані інший користувачів, що в свою чергу робить мій додаток кращим для роботи в команді

Серед недоліків маємо:

1. Менша кількість функціоналу

3.2 Порівняння з додатком Simple Time Tracker

Додаток Simple Time Tracker [24] досить схожий по функціоналу на мій застосунок, тому розглянемо інтерфейс(рис. 3.5, рис. 3.6, рис. 3.7): для того щоб знайти відмінності та виділити переваги мого застосунку.

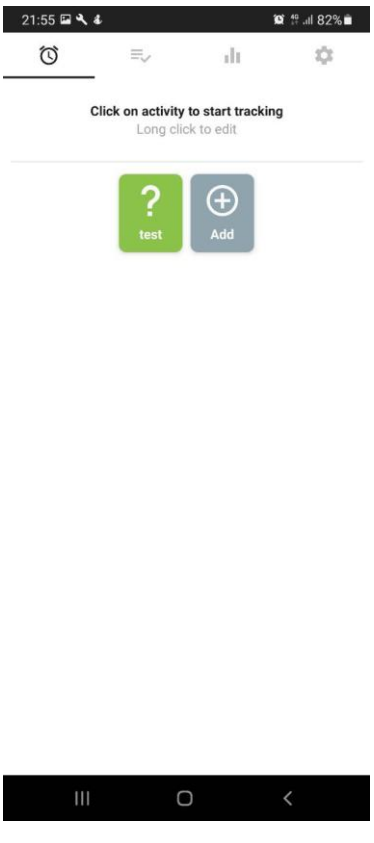

Рисунок 3.5 − Simple Time Tracker dashboard

Рисунок 3.6 − Simple Time Tracker tracked time

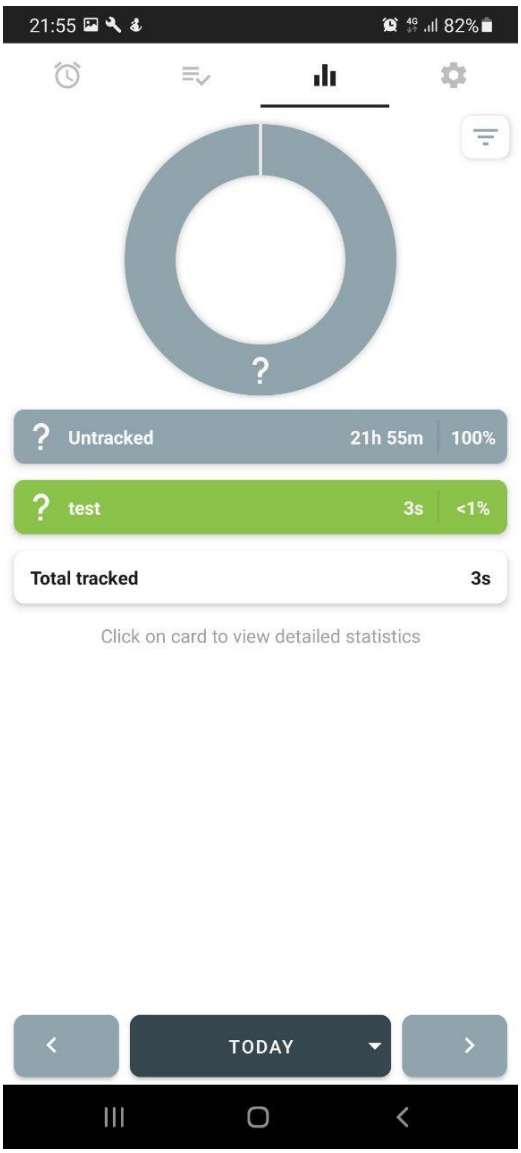

Рисунок 3.7 − Simple Time Tracker stats

Переглянувши інтерфейс даного застосунку можна зробити висновки щодо переваг на недоліків мого застосунку порівняно з Simple Time Tracker.

Переваги:

1. Функціонал зосереджений на роботу в команді
2. Можливість уточнення деталей для активностей
3. Існування альтернативної веб версії

Недоліки:

1. Відсутність функціоналу для відображення статистики
2. Відсутність функціоналу внесення даних про частину потраченого часу

3.3 Порівняння з додатком Jira

Jira[25] дуже популярний веб застосунок можливо найбільш популярний серед застосунків спрямованих саме на роботу в команді.

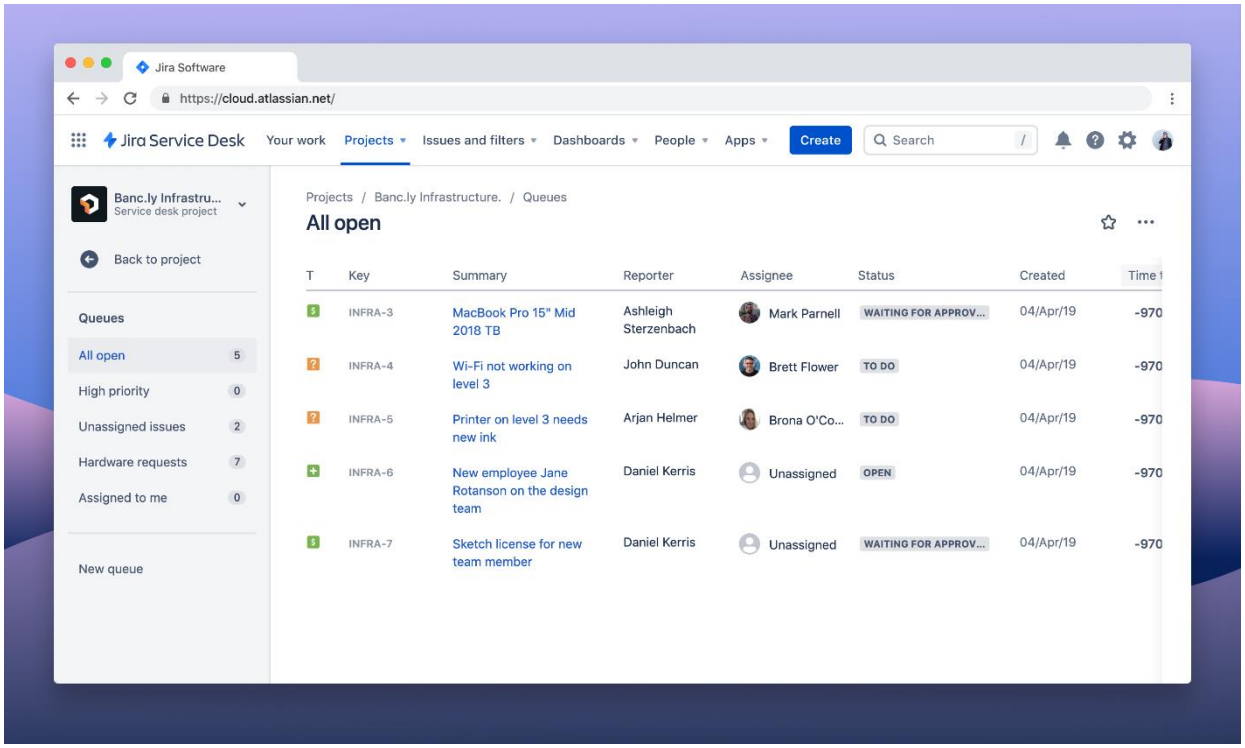

Рисунок 3.8 – Jira tasks interface

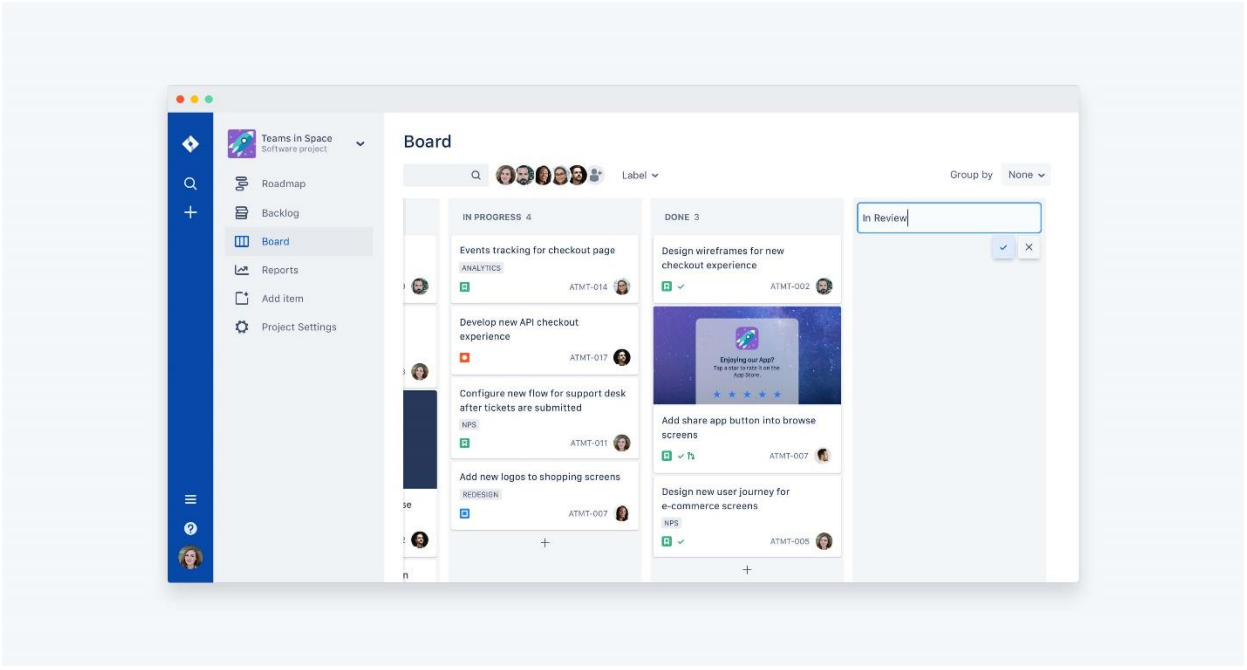

Рисунок 3.9 – Jira board interface

Загалом порівнюючи Jira(рис. 3.8, рис. 3.9) та мій веб застосунок важко

виділити багато переваг, Jira також має мобільний додаток який доповнює веб версію і весь і краще функціонал який присутній в моєму додатку, тому єдиною перевагою можна виділити простоту мого застосунку порівняно з Jira та те що користувачу легше пристосуватися до роботи з ним.

4. ДІЇ НАД ПРОЕКТОМ ПІСЛЯ РОЗРОБКИ

4.1 Тести та continuous integration

Для забезпечення правильної роботи програми та спрощення подальшої роботи над нею були написані юніт тести в обох частинах додатку. Основний функціонал тестів в серверній частині це переконатися в правильності роботи контролерів. Для цього було використана популярна бібліотека для фреймворку RoR: rspec-rails[26].

Розглянемо приклад де тестується правильність роботи аутентифікації(рис. 4.1):

```ruby
RSpec.describe 'POST /api/v1/auth', type: :request do
  let(:url) { '/api/v1/auth' }
  let(:params) do
    {
      email: "test@gmail.com",
      password: "password"
    }
  end
  let(:wrong_params) do
    {
      email: "test@gmail.com",
      password: "1password"
    }
  end

  context 'when params are correct' do
    before do
      if(User.find_by(email: params[:email]).nil?)
        Fabricate(:user)
      end
      post url, params: params
    end

    it 'returns 200' do
      expect(response).to have_http_status(200)
    end
    it 'returns JTW token ' do
      expect(response.body['token']).to be_present
    end
  end

  context 'when login params are incorrect' do
    before do
      if(User.find_by(email: params[:email]).nil?)
        Fabricate(:user)
      end
      post url, params: wrong_params
    end
```

Рисунок 4.1 − Демонстрація тесту на сервері

Для клієнтської частини була використана бібліотека jest. В основному перевірялося правильність рендерингу відповідних компонент та деяких функцій.

Але щоб постійно не запускати ці тести вручну була підключена continuous integration за допомогою Circleci[27]. Це досить зручний в користуванні сайт який буде запускати тести кожен раз коли в репозиторій з кодом робиться коміт. Але на цьому зручності не припиняються. Для перегляду результатів цих тестів достатньо відкрити Readme файл в відповідному репозиторії (рис. 4.2):

Рисунок 4.2 – Результати тестування в Readme файлі

4.2 Розгортання на сервері Heroku

В результаті маємо застосунок який може бути використаний тільки локально що не має сенсу для такого типу застосунків. Тому клієнтська[28] на серверна[29] частина була розгорнута на окремому сервері Heroku[30].

Для цього було використано відповідну офіціальну документацію[31].

Після цих дій хто завгодно може отримати доступ до мого застосунку, що дозволяє його реальне використання.

Для доступу до панелі суперадміністратора за посиланням можна використати відповідні:

email: admin@email.compassword: password

Для доступу до аккаунту з роллю розробник за посиланням можна використати відповідні:

email: gladkyy.an@gmail.compassword: password

Для доступу до аккаунту з роллю advsy за посиланням можна використати відповідні:

email: admin@gmail.compassword: password

ВИСНОВОК

У моїй дипломній роботі розроблено клієнт-серверний та мобільний застосунокавтоматизованої системи обліку та аудиту робочого часу.

1. Створено клієнт серверний застосунок.
2. Створено мобільний застосунок
3. Здійснено моделювання та розробка бази даних.
4. Впроваджено та реалізовано найбільш ефективні способи передачі даних між клієнтом та сервером;
5. Розроблено панель адміністратора для швидкого доступу до всіх даних та їх модифікації.
6. Здійснено розгортання додатку на сервері Heroku для вільного доступу до нього.
7. Додано continuous integration для забезпечення постійної перевірки коректності програми.

Отже, виконання дипломної роботи допомогло мені засвоїти теорію та отримати більше практичних навичок у використанні основ об’єктно-орієнтованого програмування та розширити власні знання у використанні сучасних мов програмування та технологій розробки клієнт-серверних та мобільних застосунків. Як кінцевий продукт було отримано клієнт-серверний та мобільний застосунок автоматизованої системи обліку та аудиту робочого часу. Який можна застосувати в багатьох сферах з великою ефективністю, хоча найбільше додаток підходить саме для компаній які займаються розробкою програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. GitHut 2.0 [Online]. Available: https://madnight.github.io/githut/#/pull_requests/2021/1.
2. Ruby on Rails [Online]. Available: https://rubyonrails.org/
3. Reenskaug, Trygve; Coplien, James O. (20 March 2009). The DCI Architecture: A New Vision of Object-Oriented Programming. Artima Developer.Retrieved 3 August 2019, ст 4
4. Action Controller Overview [Online]. Available: https://guides.rubyonrails.org/action_controller_overview.html
5. Rails Routing from the Outside In [Online]. Available: https://guides.rubyonrails.org/routing.html
6. Single-page application vs. multiple-page application [Online]. Available: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58.
7. React. JavaScript-бібліотека для створення користувацьких інтерфейсів[Online]. Available: https://uk.reactjs.org/.
8. Stack Overflow Developer Survey 2020 [Online]. Available: https://insights.stackoverflow.com/survey/2020#most-popular-technologies
9. Angular [Online]. Available: https://angular.io/
10. Stack Overflow Developer Survey 2021 [Online]. Available: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks
11. Document Object Model (DOM): definition, structure and example. IONOS Digitalguide. 2022-04-21. ст 2
12. Flutter [Online]. Available https://flutter.dev/
13. Stack Overflow Developer Survey 2021 [Online]. Available: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-other-frameworks-and-libraries

14. React-Native and Native comparison, Walmart Global Teach Blog [Online]. Available:https://medium.com/walmartglobaltech/native-vs-cross-platform-322e9896e745

15. heartcombo/devise [Online]. Available:https://github.com/heartcombo/devise

16. JSON Web Tokens [Online]. Available: https://jwt.io/

17. Mikheev, Oleg (28 August 2007). Ajax programming with Struts 2. JavaWorld. 26 July 2020. ст 1

18. Бібліотека для тестування enzyme [Online]. Available: https://enzymejs.github.io/enzyme/

19. Бібліотека для виконання HTTP запитів [Online]. Available: https://www.npmjs.com/package/axios

20. React hooks technology [Online]. Available: https://reactjs.org/docs/hooks-intro.html

21. React hook useCallback [Online]. Available: https://reactjs.org/docs/hooks-reference.html#usecallback

22. Бібліотека для взаємодії з async storage [Online]. Available: https://www.npmjs.com/package/@react-native-async-storage/async-storage

23. Work Log Tracker App [Online]. Available: https://play.google.com/store/apps/details?id=arproductions.andrew.worklog&hl=en&gl=US

24. Simple Time Tracker App [Online]. Available: https://play.google.com/store/apps/details?id=com.razeeman.util.simpletimetracker&hl=en&gl=US

25. Jira [Online]. Available: https://www.atlassian.com/software/jira

26. rspec/rspec-rails [Online]. Available: https://github.com/rspec/rspec-rails

27. Continuous Integration and Delivery - Circleci [Online]. Available: https://circleci.com/

28. activemind [Online]. Available: https://activemind.herokuapp.com/

29. activemind-api [Online]. Available: https://activemind-api.herokuapp.com/

30. Cloud Application Platform | Heroku [Online]. Available:

https://www.heroku.com/home

31. Deployment | Heroku Dev Center
https://devcenter.heroku.com/categories/deployment