ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

***«*САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**ПЕТРА ВЕЛИКОГО»**

**Институт компьютерных наук и кибербезопасности**

|  |
| --- |
|  |

**Отчет о прохождении учебной (научно-исследовательская работа (получение первичных навыков научно-исследовательской работы)) практики**

|  |
| --- |
| Кудрявцев Александр Александрович |

*(Ф.И.О. обучающегося)*

|  |
| --- |
| 1 курс, 5130203/30002 |

*(номер курса обучения и учебной группы)*

|  |
| --- |
| 02.03.03 Математическое обеспечение и администрирование информационных систем |

*(направление подготовки (код и наименование))*

|  |
| --- |
| **Место прохождения практики:** ФГАОУ ВО «СПбПУ», ИКНиК, ВШТИИ, |

*(указывается наименование профильной организации или наименование структурного подразделения*

|  |
| --- |
| г. Санкт-Петербург, ул. Обручевых, д. 1, лит. В |

*ФГАОУ ВО «СПбПУ», фактический адрес)*

|  |  |
| --- | --- |
| **Сроки практики:** с20.06.2024 по 17.07.2024 | |
| **Руководитель практической подготовки от ФГАОУ ВО «СПбПУ»:** Пак Вадим Геннадьевич, к.ф.-м.н., доцент ВШТИИ | | | |
| **(***Ф.И.О., уч. степень, должность)* | | | |
| **Йовановски Ненад** | | | |
| **(***Ф.И.О., уч. степень, должность)* | | | |
| **Руководитель практической подготовки от профильной организации**: нет | | | |
| **Оценка: 5** |
| Руководитель практической подготовки  от ФГАОУ ВО «СПбПУ»: /Пак В.Г./ | | |
| Консультант практической подготовки  от ФГАОУ ВО «СПбПУ»: /Йовановски Ненад/ | | |
| Руководитель практической подготовки  от профильной организации: /нет/ | | |
| Обучающийся: /Кудрявцев А.А./ | | |
| Дата: 16.07.24 |

**Тема**

**«Сервис для поступления в ВУЗ»**

по дисциплине «Основы программирования и алгоритмизации»

направление 02.03.03 — «Математическое обеспечение и администрирование информационных систем»

Выполнили студенты гр. 5130203/30002

Кудрявцев А.А и Мураев А.А.

Руководитель: Йовановски Ненад

«19» мая 2024 г.

Санкт-Петербург

2024

**Содержание**

Глава 1. Тема приложения и его ключевая задача…………………………………………….4

1.1. Введение…………………………………………………………………………………….4

1.2. Тема приложения………………………………………………………………..…………4

1.3. Основная цель приложения……………………………………………………………….4

Глава 2. Основные инструменты и функции приложения……………………..……………5

2.1. Мобильное приложение для абитуриентов (Кудрявцев А.)…………………………….5

2.2. Панель администратора (Мураев А.)………………………………………………….….5

2.3. Серверная часть (Кудрявцев А.)…………………………………………………………..6

Глава 3. Результаты……………………………………………………………………………..7

3.1. Репозитории с исходным кодом …………………………………………..………………7

3.2. Мобильное приложение (Кудрявцев А.А.)………………………………………………..7

3.2.1 Дизайн приложения……………………………………………………………..………..17

3.3. Сервер (Кудрявцев А.А.)…………………………………………………..………………18

3.4. Панель администратора (Мураев А.А.)…………………………………………………..32

3.4.1 Страница авторизации……………………………………………………………..……..33

3.4.2 Навигационное меню……………………………………………………………………..34

3.4.3 Главная страница – Таблица для редактирования направлений………………..……..35

3.5 Тестирование связи между сервером и панелью администраторов…………………………………………………………………………..……37

Заключение……………………………………………………………….……………………..39

Приложение 1. Математическое приложение: алгоритм Гейла-Шепли…………………40

**Глава 1. Тема приложения и его ключевая задача**

* 1. ***Введение***

Подача заявлений в университет и их обработка – это сложный процесс как со стороны приемной комиссии, так и со стороны абитуриента. Абитуриенту надо разобраться во многих сложных терминах, формировании заявки и поиске всей необходимой информации об университете. Приемной же комиссии надо вовремя все обрабатывать и отвечать на вопросы абитуриентов.

В следствие этого появилась большая потребность в автоматизации и упрощении этих процессов. Свой вариант решения этих проблем мы и представили в этой научно-исследовательской работе.

***1.2 Тема приложения***

Тема приложения является создание приложения для абитуриентов и приемной комиссии. Мы назвали его PolyAbit. **PolyAbit** (сокращенное английское Polytechnic-Abiturient) – это сервис для поступления в Политех.

* 1. ***Основная цель приложения***

Цель приложения:

1. Создать удобное мобильное приложение с доступным дизайном для абитуриентов
2. Создать функциональную панель администратора для членов приемной комиссии
3. Создать надежную серверную часть. Надежность сервера заключается в стабильности и быстроте его работы.

**Глава 2. Основные инструменты и функции приложения**

***2.1 Мобильное приложение для абитуриентов***

Для абитуриентов имеет смысл создать мобильное приложение. Все дело в том, что у большинства современных людей телефон всегда с собой, поэтому им будет удобнее через него подавать заявку и отслеживать свое положение. Кроме того, мобильные приложения обладают технологией push-уведомлений, благодаря которым он сможет быстро узнавать новости.

Мобильное приложение будет иметь следующие функционалы:

1. Получение информации о направлениях
2. Редактирование информации о себе, загрузка нужных документов

Основные инструменты:

1. Язык программирования – JavaScript, c препроцессором TypeScript
2. Фреймворк для создания мобильного приложения – Expo
3. Библиотека для роутинга – expo-router
4. Библиотека для анимаций – react-native-reanimated

***2.2 Панель администратора для абитуриентов***

Для членов приемной комиссии надо создать панель администратора в виде веб-приложения. Создание панелей администратора в форме веб-приложения – это общепринятая практика, в виду того, что сайт легче, быстрее и дешевле делать. Кроме того, основная работа членов комиссии и без того происходит в компьютере, поэтому вынесение в отдельное приложение только испортит их работу.

Панель администратора будет иметь следующие функционалы для работы с данными и пользователями.

Основные инструменты:

1. Язык программирования – JavaScript
2. Фреймворк – React
3. UI-библиотека – Bootstrap

***2.3 Серверная часть***

Сервер реализует в себе всю бизнес-логику приложения и хранения необходимых данных. Он будет связывать администраторов и абитуриентов.

Функционалы:

1. Регистрация пользователей
2. Аутентификации и авторизация пользователей
3. Обработка данных

Кроме того, сервер должен отвечать некоторым запросам:

1. Быстрота обработки запросов
2. Стабильность работы
3. Масштабируемость в часовые пики

Основные инструменты:

1. Язык программирования – Golang
2. Протокол для взаимодействия между сервисами - gRPC
3. Плагин для генерации прокси сервера: gRPC Gateway
4. База данных – SQLite 3 версии

**Глава 3.  Результаты**

***3.1 Репозитории с исходным кодом:***

Поскольку был разработан полноценный сервис, хранить весь в одном репозитории не будет правильным. За место это был создан аккаунт с репозиториями:

<https://github.com/PolyAbit>

Со следующими репозиториями:

1. docs - репозиторий, содержащий дизайн документы приложений, пользовательские соглашения и т.п.
2. admin - репозиторий с панелью администрирования
3. mobie\_app - репозиторий с мобильным приложением
4. protos - репозиторий с gRPC-контрактами
5. auth – репозиторий, содержащий сервис аутентификации и выдачи прав
6. content – репозиторий с сервисом, реализующем бизнес-логику

***3.1 Мобильное приложение (Кудрявцев А.А):***

На данный момент повсеместно используются две операционные системы для мобильных устройств: Android и IOS. Эти операционные системы значительно различаются между собой, поэтому каждая из них предлагает разработчикам свои языки программирования для создания приложений. На Android – это Kotlin, реже Java, IOS – это Swift.

Создание приложений на нативных языках для операционных систем – это хорошая практика, позволяющая создать оптимизированное приложение. Такие приложения обладают приятной пользователю плавностью пользования и быстротой работы.

Однако эта практика плоха тем, что разработка усложняется и удорожается как минимум в два раза. Все дело в том, что приложение, написанное на языке Kotlin невозможно запустить на IOS, а наоборот соответственно. Решением этой проблемы будет использование технологий кроссплатформенной разработки.

Такой технологий является React Native. React Native – это фреймворк, которая позволяет написать приложение на языке JavaScript, а запустить уже и на Android, и на IOS.

Кроссплатформенностью достигается за счет Bridge. React Native создает новые абстракции для разработчиков, при помощи которых они создают приложение. Затем уже Bridge преобразует эти абстракции в код нативных языков. Весь же JS код обворачивается в прослойку, благодаря которому нативный код можем к нему обращаться.

Благодаря React Native упростилась реализаций мобильных приложений. Однако его использование приводит к проблемам с эффективностью приложения, а также к проблемам с их поддержки. Например, появляются сложно или вообще не разрешимые баги.

React Native приобрел популярность за счет того, что построен вокруг React – фреймворка для фронтенд разработки. Собственно говоря, вокруг JSX элементов и строятся абстракции React Native (JSX Element – это «строительный блок» в React приложении).

Разберем транслирование кода на примере div – самого популярного html-тега:

То есть программист использует в реализации компонент View, зная, что это аналог тега div из HTML. Дальше уже Bridge транслирует View в нужный платформе компонент.

Однако React Native не полностью избавляет разработчиков от работы с нативными инструментами. Для запуска приложения и его финальной сборки необходимо установить программы Android Studio и XCode и их дополнительно настраивать. В случае второй будет еще одна проблема – XСode работает только на MacOS. То есть программисту надо будет обязательно купить один из ноутбуков/компьютеров Apple для полноценной работы с React Native.

Эту проблему решает фреймворк Expo – это фреймворк поверх фреймворка React Native. Он значительно упрощает разработку тем, что собирает финальные сборки на облаке, поэтому программисту не нужен будет XCode. Кроме того, Expo представляет свои библиотеки для более простой и удобной разработки. Именно поэтому был выбран Expo.

Expo поддерживает только JavaScript. JavaScript – очень популярен, потому что является стандартным языком браузерных приложений. Однако он и не лишен ряда недостатков. Одним из них является динамическая типизация. Динамическая типизация позволяет быстрее писать код, однако в долгосрочной перспективе создает большие проблемы с поддержкой кода. Во-первых, разработчиков не будет подсказки типов в редакторах кода. Во-вторых, динамическая типизация неизбежно приведет к багам в приложении.

Решением данной ситуации было создание препроцессора над JavaScript – TypeScript. TypeScript полностью совместим с JavaScript, однако добавляет в него систему типов.

**Примечание**: TypeScript зачастую называют языком, хотя в действительно это все равно препроцессор. Однако здесь и дальше авторы работы будут называть его языком.

Итак, в качестве инструментов был выбран TypeScript и Expo. Рассмотрим теперь само приложение, а точнее его структуру:

|  |  |
| --- | --- |
| - .expo  - …  - README.md  - app  - (tabs)  - \_layout.tsx  - about-app.tsx  - index.tsx  - +html.tsx  - +not-found.tsx  - \_layout.tsx  - profile  - directions.tsx  - personal-data.tsx  - sign-in.tsx  - app.json  - assets  - fonts  - PTSans-Regular.ttf  - images  - …  - babel.config.js  - components  - AnimatedEmoji.tsx  - ScreenLayout.tsx  - …  - constants  - Agreements.ts  - Api.ts  - Colors.ts  - Emoji.ts  - contexts  - auth.tsx  - expo-env.d.ts  - hooks  - useColorScheme.ts  - useStorageState.ts  - …  - package-lock.json  - package.json  - scripts  - …  - services  - http.ts  - tsconfig.json  - types  - … | Папка app содержит скрины (в мобильной разработке страницы называют скринами). Библиотека expo-router предлагает файловую систему роутинга (как в Next.JS). Благодаря этому не указываются пути напрямую в конфигурационных файлах. Кроме того, expo-router очень удобно использоваться в связке с Deep Linking – переходу по ссылке сразу в нужный раздел приложения  assets – папка со статичными материалами: шрифт, логотипы и т.п  components – UI-компоненты приложения, которые повторяются между скринами.  constants – в эту папку положили конфигурационные файлы и все константные данные  contexts, hooks, services – необходимые сущности нашего приложения  types – описания типов, встречающих в нашем приложении |

В приложении реализована аутентификации при помощи JWT-токенов. Скрин, на котором пользователи входят в приложение или регистрируются в нем называется sign-in.tsx:

**export** **default** **function** SignIn() {

**const** { signIn } = useSession();

**const** handleLogin = async () => {

**const** response = await fetch(`${Api.Auth}/login`, {

method: "POST",

body: JSON.stringify({ email, password }),

});

**if** (response.status === **200**) {

**const** { token } = await response.json();

signIn(token);

router.replace("/");

} **else** {

**const** { message } = await response.json();

// вывод сообщений об ошибке

}

};

**const** handleRegistration = async () => {

**const** response = await fetch(`${Api.Auth}/registrate`, {

method: "POST",

body: JSON.stringify({ email, password }),

});

… // вывод сообщений пользователю

};

**return** (

// верстка

);

}

В этом компоненте по нажатию на кнопку «Войти» отправляется запрос на auth-сервис приложения. В ответ он присылает сам JWT-токен, который и записывается в локальное хранилище приложения. Сама запись реализована в хуке useSession. Реализован он изнутри следующим образом:

**export** **function** useSession() {

**const** value = useContext(AuthContext);

// некоторая обработка ошибок

**return** value;

}

Этот хук был создан, чтобы код не дублировать с обраткой потенциальной ошибкой.

Рассмотрим сам контекст:

**const** AuthContext = createContext<{

signIn: (token: string) => **void**;

signOut: () => **void**;

session?: string | **null**;

isLoading: **boolean**;

}>({

signIn: (token: string) => **null**,

signOut: () => **null**,

session: **null**,

isLoading: **false**,

});

**export** **function** SessionProvider({ children }: PropsWithChildren) {

**const** [[isLoading, session], setSession] = useStorageState("session");

**return** (

<AuthContext.Provider

value={{

signIn: (token) => {

setSession(token);

},

signOut: () => {

setSession(**null**);

},

session,

isLoading,

}}

>

{children}

</AuthContext.Provider>

);

}

Контекст – это прослойка, предоставляющая реализациям все необходимые методы для работы с авторизацией пользователя: методы логина и выхода из приложения, а также сам JWT-токен. Стоит обратить внимание на использование хука useStorageState, который в свою очередь уже реализует логику работы с локальным состоянием по «ключу»:

**export** **function** useStorageState(key: string): UseStateHook<string> {

// Public

**const** [state, setState] = useAsyncState<string>();

// Get

React.useEffect(() => {

**if** (Platform.OS === "web") {

**try** {

**if** (**typeof** localStorage !== "undefined") {

setState(localStorage.getItem(key));

}

} **catch** (e) {

console.error("Local storage is unavailable:", e);

}

} **else** {

SecureStore.getItemAsync(key).then((value) => {

setState(value);

});

}

}, [key]);

// Set

**const** setValue = React.useCallback(

(value: string | **null**) => {

setState(value);

setStorageItemAsync(key, value);

},

[key]

);

**return** [state, setValue];

}

Сам хук возвращает переменную и функцию для ее изменения. Такое поведение называется иммутабельность – одну из базовых концепций React. Не будем останавливаться на этом, рекомендуем ознакомиться со статьями нашего соотечественника и разработчика React – Дэна Абрамова.

Функция изменения переменной обвернута в useCallback благодаря чему не будут лишние рендеры. Само взаимодействие с хранилищем реализовано при помощи библиотеки expo-secure-store

Использование хуков считается хорошей практикой в frontend разработке. Посмотрим на их использование в корневой странице проекта index.tsx:

**export** **default** **function** HomeScreen() {

**const** { session, isLoading } = useSession();

**if** (isLoading) {

**return** <ActivityIndicator size="large" />;

}

**if** (!session) {

**return** <Redirect href="/sign-in" />;

}

**return** (

… // верстка

);

}

Через хук в корневом скрине приложения определяться, надо ли перенаправлять пользователя на страницу с аутентификацией.

Теперь рассмотрим работу с «защищенными» методами. Для этого имеет смысл написать свой интерцептор – оболочку, который будет управлять содержимым запросов, в нашем случае добавлять JWT-токен:

**function** http<T>(

url: string,

options: FetchOptions = {}

): Promise<T | FetchError> {

**const** fullUrl = **new** URL(url, baseUrl);

**let** modifiedOptions = {

headers: {

Authorization: `Bearer ${getItem("session")}`,

"Content-Type": "application/json",

},

...options,

};

**return** fetch(fullUrl, modifiedOptions)

.then((response) => {

**if** (response.status === **401**) {

… // выход из приложения

}

**if** (!response.ok) {

**return** Promise.reject(response);

}

**return** response.json() as Promise<T>;

})

.**catch**((error) => {

console.error(error);

**throw** error;

});

}

В отличие от большинства приложений здесь используется нативное fetch API. Это было сделано для быстроты приложения. Самый известный аналог – axios, хотя и предоставляет более удобные методы, является устаревшим, поскольку внутри себя использует формат XML, из-за которого приложения работают медленнее. С другой стороны, платим красотой реализации.

Теперь в приложении есть все необходимое для работы с контентом. Рассмотрим, например, редактирование личной информации:

**export** **default** **function** PersonalData() {

**const** [firstName, setFirstName] = useState("");

**const** [lastName, setLastName] = useState("");

**const** [middleName, setMiddleName] = useState("");

**const** [loading, setLoading] = useState(**false**);

**const** fetchProfile = async () => {

setLoading(**true**);

**const** response = await http<ProfileType>("/profile", { method: "GET" });

**if** ("userId" **in** response) {

**const** profile: ProfileType = response;

setFirstName(profile.firstName);

setMiddleName(profile.middleName);

setLastName(profile.lastName);

} **else** {

**const** error: FetchError = response;

Alert.alert("Ошибка", error.error);

}

setLoading(**false**);

};

**const** handleSave = async () => {

setLoading(**true**);

**const** response = await http<ProfileType>("/profile", {

method: "PUT",

body: JSON.stringify({ firstName, lastName, middleName }),

});

**if** ("userId" **in** response) {

Alert.alert("Успешно", "Профиль успешно обновлен");

} **else** {

**const** error: FetchError = response;

Alert.alert("Ошибка", error.error);

}

setLoading(**false**);

};

useEffect(() => {

fetchProfile();

}, []);

**return** (

.. // верстка

);

}

Здесь реализован метод для получения профиля и его редактирования.

***3.1.1 Дизайн приложения***

Дизайн, предназначенный для конечных клиентов, не может быть чем-то утилитарным. Его следует делать максимально приятным и простым в использовании. Дизайн должен расставить акценты, благодаря которым пользователь быстро разберется в приложении.

Таким критерием отвечает философия «человеческого» дизайна от Apple. Он представляет собой манифест с общими правилами создания интерфейсов, понятных людям. Полностью манифест можно прочитать по этой ссылке <https://developer.apple.com/design/human-interface-guidelines>. Референс дизайна PolyAbit – это Apple Музыка.

Кроме того, в приложение надо было показать принадлежность к экосистеме Политеха. Для этого была прочитана методичка с описанием фирменного стиля - <https://www.spbstu.ru/upload/branding/guide-2022.pdf>. Для этого в приложении использовался фирменный зеленый цвет и шрифт.

Результаты представлены ниже:

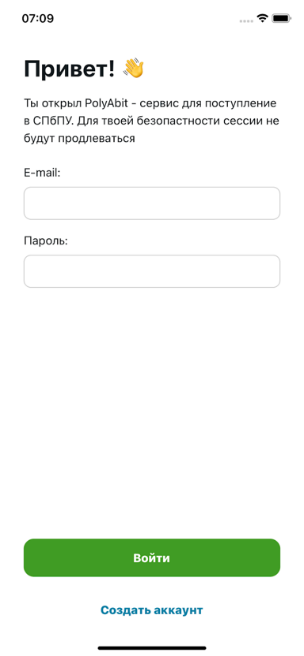
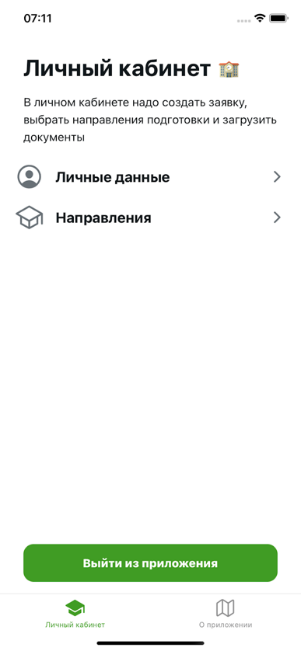
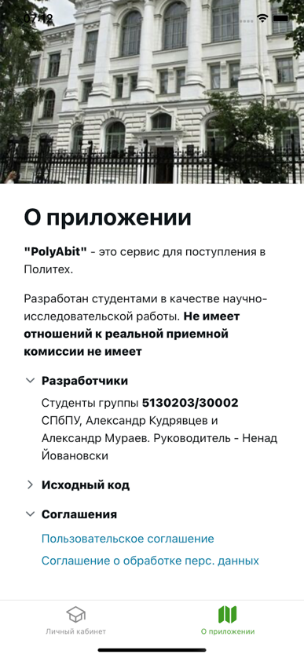
  ****

Рис 1.1 – скрин с авторизацией

Рис 1.2– скрин с личным кабинетом

Рис 1.3 – скрин с разделом «О приложении»

Особое внимание стоит обратить на описание страниц. Для них реализована анимация параллакса, чтобы у пользователей была плавность при листании. И самое интересное, анимированы иконки, чтобы пользователь при открытии страницы сразу обратил внимание на текст.

***3.2 Сервер (Кудрявцев А.):***

Язык Go – это современный язык программирования от Google, призванный прийти на замену устаревшим Java и .Net и крайне медленным Python и Ruby. Язык Go был спроектирован так, чтобы он обладал всеми преимуществами высокоуровневого программирования, при этом не теряя в производительности. Go нашел широкое применение в серверной разработке в виду его быстроты и доступности в изучении.

Особенно часто Go используется в микросервисной архитектуру. Она заключается в том, что приложение делится на приложение поменьше, в которых уже непосредственно реализована вся бизнес-логика.

Изначально сервера разрабатывались по монолитной архитектуру, где в рамках одного приложения была вся логика. Такие приложения легко писать и разворачивать на облаке, но оно обладает рядом недостатков.

Во-первых, при разрастании проекта его станет сложно поддерживать. Использование подходов по типу модульной или луковой архитектуры (к сожалению, здесь тоже используется слово архитектуру, но его стоит понимать в другом значении, более прикладном) позволит избежать вертикального роста проекта, но никак не горизонтального.

Во-вторых, такие проекты сложно масштабировать. Разберем на примере ситуации, когда абитуриенты судорожно смотрят на списки по несколько десятков раз в день каждый. Зачастую сервера университетов в это время падают, что было, например, в нашем университете во время приемной комиссии 2023 года. Использование микросервисной архитектуры позволяет обойти эту проблему. В данной ситуации технические специалисты могут просто масштабировать сервис, формирующий конкурсные списки, например, в 10 раз и он стабильно отработает.

К сожалению, использование микросервисов влечет к новым проблемам. Первая, это сложность разворачивания сервера на облаке. Эта тема не будет раскрываться в научной работе в виду того, что эта тема как минимум достойно еще одной полноценной работы. Вторая, это коммуникация сервисов между собой.

Коммуникации между разделить на синхронные и асинхронные. Первые предполагают получение ответа напрямую, а вторые – нет. Например, получение прав пользователя – это синхронная коммуникация, а обращение к mail сервису – это асинхронная.

Примеры инструментов для синхронной: протоколы http, soap или современный gRPC. Для асинхронных коммуникаций используется брокеры сообщений по типу Kafka. К сожалению, брокеры сообщений – это тема очень глубокая, изучению которой тоже можно посвятить отдельную работу. В нашем примере ограничились gRPC.

Протокол gRPC – это новый протокол от Google, реализованный поверх http 2.0. Он изначально был спроектирован для общения сервисов между собой. Это достигается и за счет вышеупомянутого http 2.0, в котором не нужно создавать для каждого запроса еще одно подключение, и за счет использования бинарных данных, заместо текстовых, по типу JSON.

Протокол gRPC быстрее доминирующего http 1.0. Например, он справляется со медиа контентом в 4 раза, а с текстовым – до 10 раз. За такую производительность приходится платить тем, что протокол gRPC не поддерживается большим количеством клиентов, в том числе и веб-приложениями. Для этого нам придется написать прокси сервер (здесь и далее будем называть его API Gateway), который бы транслировал запросы из http в gRPC формат.

Рассмотрим использование gRPC уже на примере нашего проекта. Протокол требует явно определения методов и их содержимых. Для этого надо прописать proto контракты.

Контракты и команды генерации кода лежат в репозитории protos.

Рассмотрим на примере контракта для сервиса content.

|  |  |
| --- | --- |
| syntax = "proto3";  **package** **content**;  **import** "google/api/annotations.proto";  **option** go\_package = "PolyAbit.content.v1;contentv1";  **service** Content {  **rpc** GetProfile(GetProfileRequest) **returns** (Profile) {  **option** (google.api.http) = {  get: "/profile"  };  }  **rpc** UpdateProfile(Profile) **returns** (Profile) {  **option** (google.api.http) = {  put: "/profile",  body: "\*"  };  }  **rpc** GetDirections(Empty) **returns** (Directions) {  **option** (google.api.http) = {  get: "/direction"  };  }  **rpc** CreateDirection(CreateDirectionRequest) **returns** (Empty) {  **option** (google.api.http) = {  post: "/direction",  body: "\*"  };  }  **rpc** DeleteDirection(DeleteDirectionRequest) **returns** (Empty) {  **option** (google.api.http) = {  delete: "/direction/{directionId}"  };  }  }  **message** **Empty** {}  **message** **Profile** {  **int64** userId = **1**;  **string** firstName = **2**;  **string** middleName = **3**;  **string** lastName = **4**;  }  **message** **Direction** {  **int64** id = **1**;  **string** code = **2**;  **string** name = **3**;  **string** exams = **4**;  **string** description = **5**;  }  … // и другие | Объявляем название пакета и версию используемого синтаксиса  Импортируем плагин, который будет генерировать уже готовый API Gateway  Задаем название, которым буду обладать уже Go пакет  Здесь описывает сам сервис и его методы: название, его параметры и ответы. Особое внимание стоит обратить на опции, потому что в них указывается, с какого адреса будет проксироваться запрос.  Дальше идет объявление всех сущностей. |

Чтобы эти протоколы заработали необходимо сгенерировать код, который реализует всю логику работы пакета. В дальнейшем разработчику надо будет просто воспользоваться сгенерованными моделями и написать свои реализации методов. Это называется кодогенерация. И ее можно запустить примерно таким образом:

protoc --proto\_path proto/content --proto\_path vendor.protogen **\**

--go\_out=gen/go/content --go\_opt=paths=source\_relative **\**

--plugin=protoc-gen-go=bin/protoc-gen-go **\**

--go-grpc\_out=gen/go/content --go-grpc\_opt=paths=source\_relative **\**

--plugin=protoc-gen-go-grpc=bin/protoc-gen-go-grpc **\**

--grpc-gateway\_out=gen/go/content --grpc-gateway\_opt=paths=source\_relative **\**

--plugin=protoc-gen-grpc-gateway=bin/protoc-gen-grpc-gateway **\**

proto/content/content.proto

Для простоты в репозитории protos есть Makefile, в котором они уже все написаны. Дальше остается просто поделиться этими сгенерированными файлами с сервисами. Это можно сделать при помощи GitHub, хорошей практикой будет сделать тэги с версиями нашего протокола.

Рассмотрим теперь сами микросервисы нашего приложения:

1. auth – это сервис, отвечающий за регистрацию пользователей, выдачи им JWT-токенов и их permissions (прав).
2. content – это сервис, реализующий логику обработки данных

Между собой они очень похожи, потому что были построены по одному и тому же принципу – DI. Dependency Injection – это принцип управления зависимостей. Согласному нему каждый слой должен работать не с зависимостями напрямую, а их с интерфейсами, и, соответственно, быть точно такой же завимостью для других слоев.

Рассмотрим это на архитектуре приложения:

|  |  |
| --- | --- |
| - .gitignore  - README.md  - Taskfile.yaml  - cmd  - content  - main.go  - migrator  - main.go  - config  - local.yaml  - go.mod  - go.sum  - internal  - app  - app.go  - grpc  - app.go  - clients  - auth  - grpc  - grpc.go  - isAdmin.go  - config  - config.go  - grpc  - content  - server.go  - lib  - … // внутренние вспомогательные файлы  - models  - direction.go  - errors.go  - profile.go  - services  - content  - content.go  - direction.go  - errors.go  - profile.go  - storage  - sqlite  - sqlite.go  - migrations  - … // миграции БД  - storage  - content.db | Точкой входа в приложение является файл cmd/content/main.go. Здесь main файл выделен в команды.  В cmd можно найти и другие команды, например, команда по запуску миграций БД  internal – вся логика приложения. В нем есть важные подпапки:  app – все мини-приложения, которые могут быть запущены, и само наше приложение. Например, gRPC сервер.  clients – реализация клиента, то есть сущностей, которые будут работать с другими сервисами. В нашем случае – это auth сервис и метод для получения прав пользоваться  config – обвертка для работы с конфигурацией приложения.  grpc – слой контроллеров для gRPC сервиса; services – слой сервисов; storage – слой для работы с БД. Все это является классическим MVC паттерном.  lib – файлы, с полезным кодом  migrations – папка со всеми миграциями БД  storage – папка с БД |

Теперь рассмотрим важные и сложные участки. Во-первых, запуск gRPC сервиса и API Gateway к нему:

**package** grpcapp

**import** (

… // все необходимые пакеты

)

**type** App **struct** {

log \*slog.Logger

gRPCServer \*grpc.Server

grpcPort **int**

httpPort **int**

}

**func** New(log \*slog.Logger, contentService grpccontent.Content, permProvider middleware.PermissionProvider, gRPCPort **int**, httpPort **int**, jwtSecret **string**) \*App {

… // инициализация middlewares

gRPCServer := grpc.NewServer(grpc.ChainUnaryInterceptor(unaryInterceptors...))

grpccontent.Register(gRPCServer, contentService)

**return** &App{

log: log,

gRPCServer: gRPCServer,

grpcPort: gRPCPort,

httpPort: httpPort,

}

}

**func** (a \*App) MustRun() {

**if** err := a.Run(); err != **nil** {

panic(err)

}

}

**func** (a \*App) Run() **error** {

**const** op = "grpcapp.Run"

ctx := context.Background()

wg := &sync.WaitGroup{}

wg.Add(**2**)

**go** **func**() {

**defer** wg.Done()

**if** err := a.startGrpcServer(); err != **nil** {

log.Error("failed start grpc", sl.Err(err))

}

}()

**go** **func**() {

**defer** wg.Done()

**if** err := a.startHttpServer(ctx); err != **nil** {

log.Error("failed start http", sl.Err(err))

}

}()

wg.Wait()

**return** **nil**

}

**func** (a \*App) startGrpcServer() **error** {

**const** op = "app.grpc.startGrpcServer"

l, err := net.Listen("tcp", fmt.Sprintf(":%d", a.grpcPort))

**if** err != **nil** {

**return** fmt.Errorf("%s: %w", op, err)

}

**if** err := a.gRPCServer.Serve(l); err != **nil** {

**return** fmt.Errorf("%s: %w", op, err)

}

**return** **nil**

}

**func** (a \*App) startHttpServer(ctx context.Context) **error** {

mux := runtime.NewServeMux()

opts := []grpc.DialOption{

grpc.WithTransportCredentials(insecure.NewCredentials()),

}

err := contentv1.RegisterContentHandlerFromEndpoint(ctx, mux, fmt.Sprintf(":%d", a.grpcPort), opts)

**if** err != **nil** {

**return** err

}

**return** http.ListenAndServe(fmt.Sprintf("localhost:%d", a.httpPort), mux)

}

Здесь мы создается структуру App, в которой мы будем хранить полученные с уровня выше зависимости. Дальше надо обратить внимание на метод Run, потому что в нем при помощи горутин – мощного механизма языка Go – создается два сервера: gRPC и API Gateway для него.

Код, как и везде выше, само собой отчасти сокращенный. Но стоит упомянуть, что для работы с API Gateway создался обработчик CORS, а для gRPC – Grateful Shutdown.

В корневом app собирается все завимости воедино:

**package** app

**import** (

"context"

"log/slog"

grpcapp "github.com/PolyAbit/content/internal/app/grpc"

grpcauth "github.com/PolyAbit/content/internal/clients/auth/grpc"

"github.com/PolyAbit/content/internal/config"

"github.com/PolyAbit/content/internal/services/content"

"github.com/PolyAbit/content/internal/storage/sqlite"

)

**type** App **struct** {

GRPCServer \*grpcapp.App

}

**func** New(

log \*slog.Logger,

cfg \*config.Config,

) \*App {

storage, err := sqlite.New(cfg.StoragePath)

**if** err != **nil** {

panic(err)

}

contentService := content.New(log, storage, storage)

authClient, err := grpcauth.New(context.Background(), log, cfg.Clients.Auth.Address, cfg.Clients.Auth.Timeout, int(cfg.Clients.Auth.RetriesCount))

**if** err != **nil** {

panic(err)

}

grpcApp := grpcapp.New(log, contentService, authClient, cfg.GRPC.Port, cfg.GRPC.GatewayPort, cfg.JwtSecret)

**return** &App{

GRPCServer: grpcApp,

}

}

Здесь создается завимость для работы с БД и бизнес-логикой – storage и contentService соответственно.

Рассмотрим реализацию клиента:

**package** grpc

**import** (

"context"

"fmt"

"log/slog"

"time"

authv1 "github.com/PolyAbit/protos/gen/go/auth"

grpclog "github.com/grpc-ecosystem/go-grpc-middleware/v2/interceptors/logging"

grpcretry "github.com/grpc-ecosystem/go-grpc-middleware/v2/interceptors/retry"

"google.golang.org/grpc"

"google.golang.org/grpc/codes"

"google.golang.org/grpc/credentials/insecure"

)

**type** Client **struct** {

api authv1.AuthClient

}

**func** New(

ctx context.Context,

log \*slog.Logger,

addr **string**,

timeout time.Duration,

retriesCount **int**,

) (\*Client, **error**) {

**const** op = "grpc.New"

… // инициализация множества middleware

cc, err := grpc.DialContext(ctx, addr, … // передача их))

**if** err != **nil** {

**return** **nil**, fmt.Errorf("%s: %w", op, err)

}

grpcClient := authv1.NewAuthClient(cc)

**return** &Client{

api: grpcClient,

}, **nil**

}

**func** (c \*Client) IsAdmin(ctx context.Context, userID **int64**) (**bool**, **error**) {

**const** op = "grpc.IsAdmin"

resp, err := c.api.IsAdmin(ctx, &authv1.IsAdminRequest{

UserId: userID,

})

…// дальнейшая обработка

}

Здесь видим, что используется сгенерированный по нашему протоколу пакет github.com/PolyAbit/protos/gen/go/auth из репозитория protos. Здесь видно, что достаточно было просто вызвать метод IsAdmin и получить результат. Кодогенерация уже создала разработчику необходимый клиент – достаточно всего лишь воспользоваться.

В дальнейшем этот клиент может понадобиться, когда надо будет получить права пользователя. Например, в auth middleware. Здесь уже не единожды использовался этот термин. Middleware – это промежуточный обработчик запроса, который нужным образом меняет запрос. В нашем случае он будет получить token пользователя, его валидировать и получать c auth сервера права:

**package** middleware

**import** (

… // много импортов

)

**type** AuthMiddleware **struct** {

AuthFunc **func**(ctx context.Context) (**bool**, context.Context, **error**)

}

**type** UserClaim **struct** {

jwt.RegisteredClaims

Uid **int64**

Email **string**

}

**type** userId **string**

**type** isAdmin **string**

**const** (

userKey userId = "userId"

isAdminKey isAdmin = "isAdmin"

)

**type** PermissionProvider **interface** {

IsAdmin(ctx context.Context, userId **int64**) (**bool**, **error**)

}

**func** (a \*AuthMiddleware) UnaryInterceptor(ctx context.Context, req **interface**{}, info \*grpc.UnaryServerInfo, handler grpc.UnaryHandler) (**interface**{}, **error**) {

authorized, updatedContext, err := a.AuthFunc(ctx)

**if** err != **nil** {

**return** **nil**, err

}

**if** !authorized {

**return** **nil**, status.Errorf(codes.Unauthenticated, "User is not authorized")

}

**return** handler(updatedContext, req)

}

**func** New(key **string**, permProvider PermissionProvider) **func**(context.Context) (**bool**, context.Context, **error**) {

**return** **func**(ctx context.Context) (**bool**, context.Context, **error**) {

md := metadata.ExtractIncoming(ctx)

authorizationHeader := md["authorization"]

**if** authorizationHeader == **nil** || authorizationHeader[**0**] == "" {

**return** **false**, ctx, **nil**

}

… // валидация токена

isAdmin, err := permProvider.IsAdmin(ctx, userClaim.Uid)

**if** err != **nil** {

**return** **false**, ctx, fmt.Errorf("failed to get permission: %w", err)

}

ctx = context.WithValue(ctx, isAdminKey, isAdmin)

**return** **true**, ctx, **nil**

}

}

Видим, что конструктор middleware принимает завимость и уже в самой обработчике вызывает нужный метод. Дальше он просто запишет в контекст данные об пользователе и его правах. Хорошей практикой будет написать следующие утилиты:

**func** UIDFromContext(ctx context.Context) (**int64**, **bool**) {

uid, ok := ctx.Value(userKey).(**int64**)

**return** uid, ok

}

**func** IsAdminFromContext(ctx context.Context) (**bool**, **bool**) {

isAdmin, ok := ctx.Value(isAdminKey).(**bool**)

**return** isAdmin, ok

}

Вернемся назад и рассмотрим реализацию репозитория:

**package** sqlite

**import** (

"context"

"database/sql"

"errors"

"fmt"

"github.com/PolyAbit/content/internal/models"

"github.com/mattn/go-sqlite3"

)

**type** Storage **struct** {

db \*sql.DB

}

**func** New(storagePath **string**) (\*Storage, **error**) {

**const** op = "storage.sqlite.New"

db, err := sql.Open("sqlite3", storagePath)

**if** err != **nil** {

**return** **nil**, fmt.Errorf("%s: %w", op, err)

}

**return** &Storage{db: db}, **nil**

}

**func** (s \*Storage) SaveDirection(ctx context.Context, code **string**, name **string**, exams **string**, description **string**) **error** {

**const** op = "storage.sqlite.SaveUser"

stmt, err := s.db.Prepare("INSERT INTO direction(code, name, exams, description) VALUES(?, ?, ?, ?)")

**if** err != **nil** {

**return** fmt.Errorf("%s: %w", op, err)

}

\_, err = stmt.ExecContext(ctx, code, name, exams, description)

**if** err != **nil** {

**var** sqliteErr sqlite3.Error

**if** errors.As(err, &sqliteErr) && sqliteErr.ExtendedCode == sqlite3.ErrConstraintUnique {

**return** fmt.Errorf("%s: %w", op, models.ErrDirectionExists)

}

**return** fmt.Errorf("%s: %w", op, err)

}

**return** **nil**

}

Здесь в конструкторе создается подключение к базе данных при помощи пакета из стандартной библиотеки Go. Дальше уже при помощи этой же библиотеки выполняется и сам запрос.

То есть в данной реализации никак не использовались ORM. Подход ORM можно считать устаревшим, потому что большинство языков дает поддержку типов и, кроме того, ORM генерируют неоптимальные запросы. ORM неплохо справляются с простыми запросами, но с простыми запросами и тем более справится разработчик.

***3.4 Панель администратора(Мураев А.):***

Панель администратора написана на React, на ванильном JavaScript. В качестве UI библиотеки использовался React Bootstrap.

Архитектура проекта очень простая, потому что является классических без архитектурным React проектом. Однако в виду ее ограниченного функционала такой подход является правильным.

WEB-клиент представляет из себя две страницы:

1. Страница авторизации
2. Главная страница с таблицей для редактирования направлений

На главной странице предусмотрен навигационный блок, который позволяет интерактивно переключаться между другими страницами, которые в будущем будут созданы.

***3.4.1 Страница авторизации***

На странице авторизации присутствует логика авторизации пользователя, пользователь предоставляет почту-логин и пароль, который отправляется на сервер для проверки, после чего пользователю выдается JWT-ТОКЕН, который прикрепляется к остальным запросам со стороны клиента, для предотвращения несанкционированного доступа к админ-панели. Ниже представлена логика запроса на авторизацию. Кнопка ‘Войти’ заблокирована, пока поля не будут заполнены корректно.

**const** handleSubmit = async (e) => {

e.preventDefault();

**try** {

**const** response = await fetch(`${Api.Auth}/login`, {

method: "POST",

body: JSON.stringify({ email, password }),

});

**const** { token } = await response.json();

**if** (token) {

sessionStorage.setItem("Token", token);

handleLogin();

}

} **catch** (error) {

console.log("Ошибка при авторизации: ", error);

}

};

Дизайн страницы:

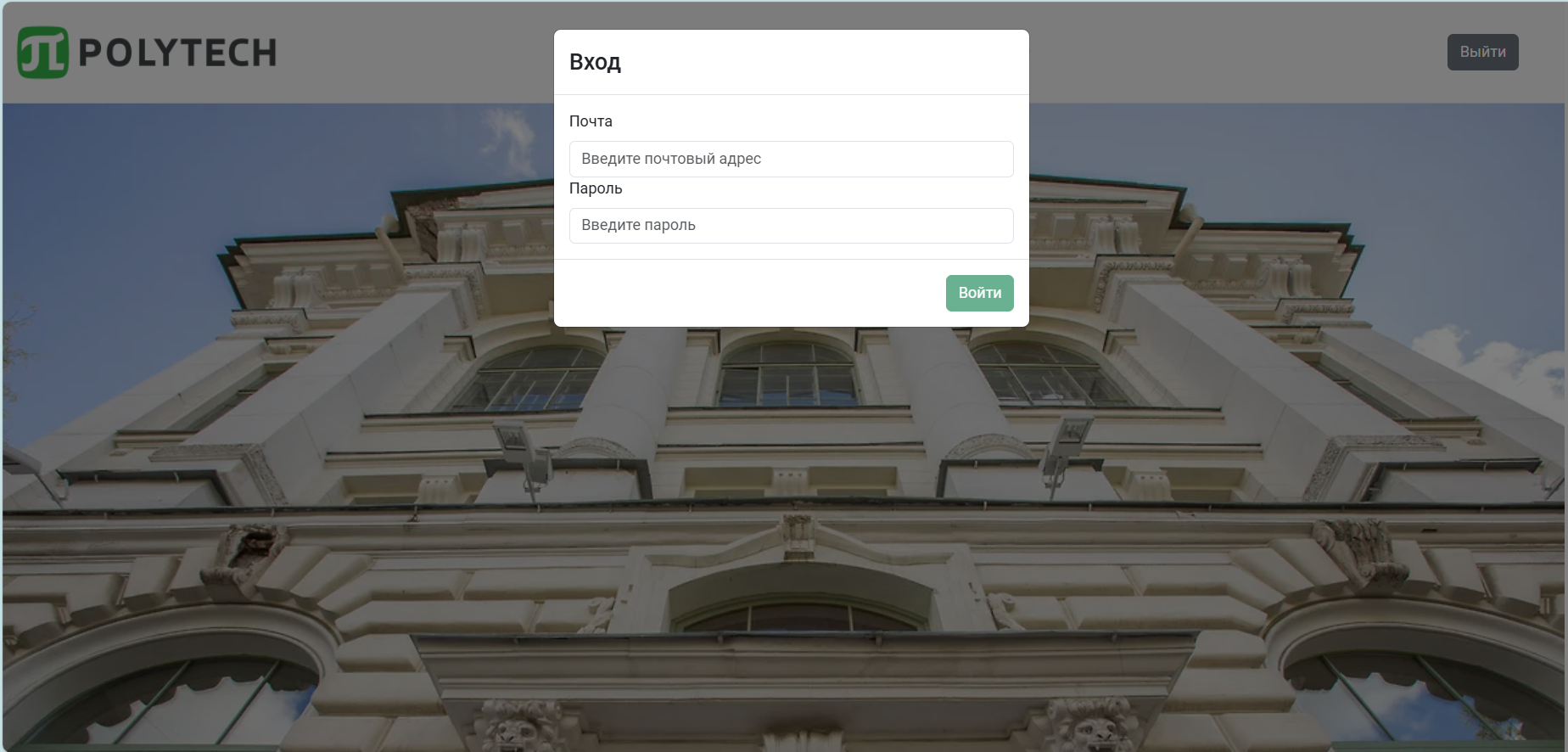


Рис.2.1. Страница авторизации для WEB-клиента администраторов

***3.4.2 Навигационное меню***

В навигационной панели есть кнопка “Выйти”, которая разлогинивает пользователя и удаляет его токен для обеспечения безопасности.

При клике на логотип происходит переадресация на главную страницу СПБПУ Петра Великого.

Панель едина для всех будущих разделов главной страницы, которые на данный момент не завершены и переход на них заблокирован.

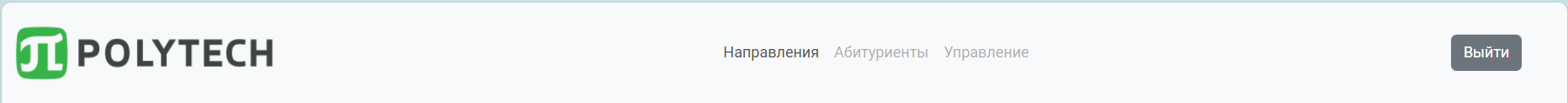
******

Рис.2.2 Навигационное меню для WEB-клиента администраторов

***3.4.3 Главная страница – Таблица для редактирования направлений***

В функционал таблицы входят:

1. Модальное окно с формой для заполнения таблицы
2. Таблица с возможностью удаления отдельных строк

При загрузке страницы отправляется запрос на сервер для вывода уже имеющейся информации по направлениям в таблицу. Метод запроса `GET`, в заголовок `authorization` прикрепляется токен доступа. Ниже представлен код.

**const** fetchDirections = async () => {

**const** response = await fetch(`${Api.Content}/direction`, {

method: "GET",

headers: {

authorization: `Bearer ${sessionStorage.getItem("Token")}`,

"content-type": "application/json",

},

});

**try** {

**if** (response.status === **200**) {

**const** { directions } = await response.json();

setDirections(directions);

} **else** {

setDirections([]);

}

} **catch** (error) {

console.log(error);

}

};

useEffect(() => {

fetchDirections();

}, []);

При нажатии на кнопку ‘Зарегистрировать направление’ вызывается модальное окно, где при нажатии на кнопку “Зарегистрировать” данные отправляются на сервер для сохранения при помощи `POST`запроса, в заголовок к которому так же прикрепляется токен доступа.

**const** handleCreateDirection = async (payload) => {

**const** response = await fetch(`${Api.Content}/direction`, {

method: "POST",

headers: {

authorization: `Bearer ${sessionStorage.getItem("Token")}`,

},

body: JSON.stringify(payload),

});

**try** {

**if** (response.ok) {

fetchDirections();

}

} **catch** (error) {

console.log(error);

}

};

При нажатии на иконку ‘Удалить’ на сервер отправляется запрос `DELETE`, так же с токеном доступа в заголовке. Данные удаляются как из таблицы администратора, так и на сервере, что позволяет избегать несоответствий в информации на сервере и у администратора. После удаления информация в таблице обновляется.

**const** handleDeleteDirection = async (id) => {

**if** (!window.confirm("Вы уверены, что хотите удалить эту строку?")) {

**return**;

}

**const** response = await fetch(`${Api.Content}/direction/${id}`, {

method: "DELETE",

headers: {

authorization: `Bearer ${sessionStorage.getItem("Token")}`,

},

});

**try** {

**if** (response.status === **200**) {

fetchDirections();

}

} **catch** (error) {

console.log(error);

}

};

***Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание***

Рис.2.3 Таблица для редактирования направлений

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Рис.2.4 Форма для заполнения таблицы направлений.

***3.5. Тестирование WEB-клиента панели администраторов***

***Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание***

Рис.3.1 Тест POST запроса на авторизацию

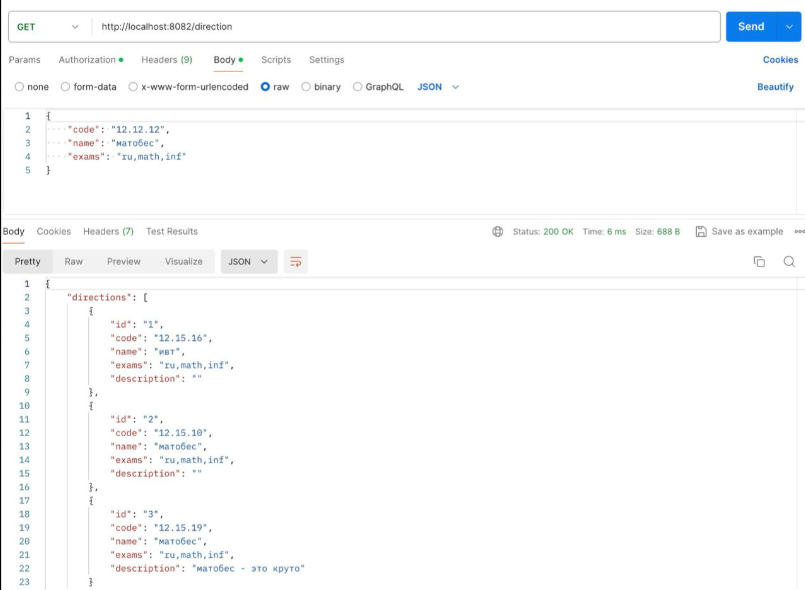
******

Рис.3.2 Тест GET запроса на получение данных с сервера

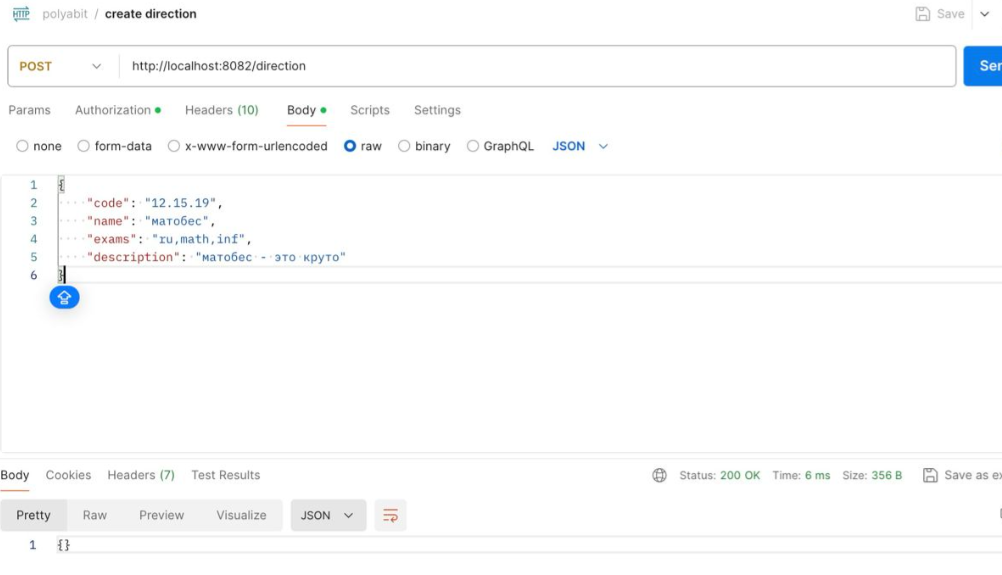
******

Рис.3.3 Тест POST запроса на отправку данных на сервер

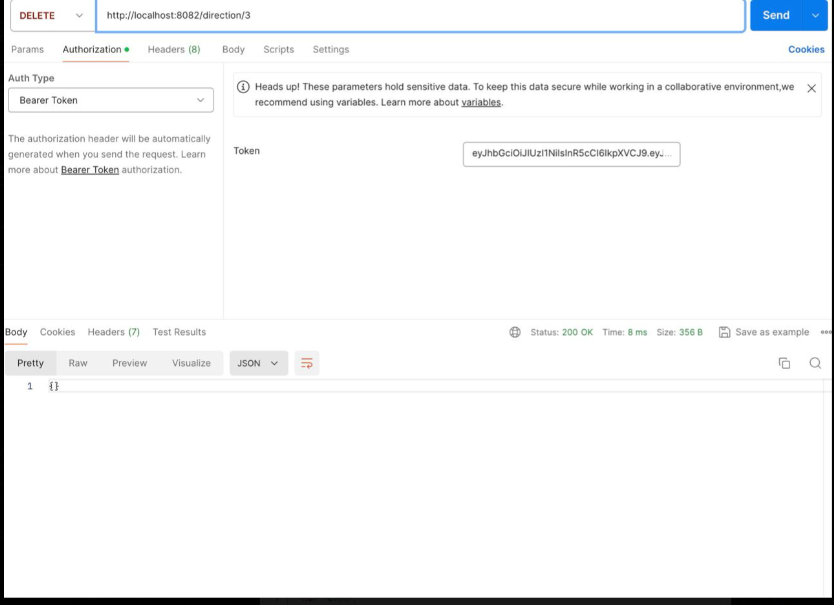


Рис.3.4 Тест DELETE запроса на удаление данных с сервера

**Заключение**

Проект по разработке платформы для абитуриентов и приемной комиссии успешно завершил свой первый этап. За прошедший период была проделана колоссальная работа, в результате которой реализованы ключевые функции, необходимые для комфортного использования сервиса.

На сегодняшний день доступны:

1. Удобное мобильное приложение с доступным дизайном для абитуриентов: приложение простое в использовании, с интуитивно понятным интерфейсом, позволяющим быстро найти необходимую информацию.
2. Функциональная панель администратора для членов приемной комиссии: панель предоставляет основные инструменты для эффективного управления данными
3. Надежная серверная часть: сервер стабилен, масштабируем, он обеспечивает быстрый доступ к платформе.

В рамках текущего этапа, также заложен фундамент для дальнейшего развития сервиса. Это дополнительно проверено тестированием всех функционалов.

Сервис будет постоянно развиваться, приобретая новые функции и возможности.

На основе этой работы в будущем могут быть написаны другие исследования. Например, можно исследовать возможности DevOps и настроить в сервисе систему CI/CD или написать еще один сервис «уведомлений», работать с которым при помощи брокеров сообщений.

**Приложения**

***Математическое приложение: алгоритм Гейла-Шепли***

Представьте, что есть несколько университетов, предлагающих разные направления, и студенты, которые хотят поступить на эти направления. Каждый студент ранжирует направления по своим предпочтениям, а каждый университет ранжирует студентов по своим критериям (например, баллы ЕГЭ).

Алгоритм Гейла-Шепли позволяет найти стабильное распределение студентов по направлениям, где:

* Ни один студент не хотел бы поменяться направлением с другим студентом.
* Ни один университет не хотел бы поменять одного студента на другого.

Как это работает:

1. Инициализация: Каждый студент делает предложение своему самому предпочтительному направлению.
2. Рассмотрение предложений: Университет рассматривает полученные предложения и принимает студентов в порядке своего ранжирования, пока не заполнит все места.
3. Отклонение: Студенты, чьи предложения были отклонены, делают предложение следующему направлению в своем списке предпочтений.
4. Повторение: Шаги 2 и 3 повторяются, пока все студенты не получат место на каком-либо направлении.

Преимущества:

* Стабильность: Алгоритм гарантирует стабильное распределение, где никто не хочет менять местами.
* Справедливость: Алгоритм учитывает предпочтения как студентов, так и университетов.
* Эффективность: Алгоритм работает быстро и эффективно.

Пример:

* Студент А хочет поступить на направление "Компьютерные науки" в университете X, а затем на "Математику" в университете Y.
* Студент B хочет поступить на "Математику" в университете Y, а затем на "Компьютерные науки" в университете X.
* Университет X предпочитает студента B, а университет Y предпочитает студента A.

Алгоритм Гейла-Шепли найдет стабильное распределение, где студент A поступит на "Математику" в университете Y, а студент B поступит на "Компьютерные науки" в университете X.

**Важно**: Алгоритм Гейла-Шепли не всегда приводит к распределению, которое было бы идеальным для каждого студента. Однако он гарантирует стабильность и справедливость в условиях ограниченного количества мест.