ОГЛАВЛЕНИЕ

[ВВЕДЕНИЕ 3](#_Toc172563020)

[ГЛАВА 1. ВСЕСТОРОНЕЕ ИЗУЧЕНИЕ ОБЪЕКТЫ И ПРЕДМЕТА 4](#_Toc172563021)

[1.1 О приложении PolyAbit 4](#_Toc172563022)

[1.2 Историческая справка 4](#_Toc172563023)

[1.3 Исследование и обзор технологий 5](#_Toc172563024)

[1.3.1 Исследование и обзор технологий для создания мобильных приложений 5](#_Toc172563025)

[1.3.2 Исследование и обзор технологий для создания серверных приложений 7](#_Toc172563026)

[1.3.3 Исследование и обзор технологий для создания веб-сайтов 9](#_Toc172563027)

[ГЛАВА 2. РЕАЛИЗАЦИЯ СОВРЕМЕННОГО ВЕБ-СЕРВИСА 10](#_Toc172563028)

[2.1 Серверное приложение 10](#_Toc172563029)

[2.2 Мобильное приложение 14](#_Toc172563030)

[ГЛАВА 3. АПРОБАЦИЯ РЕЗУЛЬТАТОВ ИССЛЕДОВАНИЯ 18](#_Toc172563031)

[3.1. Введение 18](#_Toc172563032)

[3.2. Тестирование запросов 18](#_Toc172563033)

[ЗАКЛЮЧЕНИЕ 20](#_Toc172563034)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 21](#_Toc172563035)

# ВВЕДЕНИЕ

В современной информационном обществе важную роль играют веб-сервисы. Они изменили многие аспекты нашей жизни. Например, для заказа такси большинство людей воспользуется каким-нибудь сервисом, например, Яндекс Go.

Такие сервисы удобны для пользователей, потому что они предоставляют интуитивно-понятный интерфейс. Кроме того, современные веб-сервисы могут одновременно обрабатывать большое количество пользователей.

В работе будет произведено изучение актуальных методов и технологий для создания современных высоконагруженных веб-сервисов.

Целью практики является исследование реализаций веб-сервисов и их применении для создания веб-сервиса приемной комиссии СПбПУ.

Для достижения цели были поставлены задачи:

1) Изучить высоконагруженные серверные приложения.

2) Изучить правила создания интуитивно-понятных интерфейсов.

3) Создать прототип веб-сервиса приемной комиссии СПбПУ.

# ГЛАВА 1. ВСЕСТОРОНЕЕ ИЗУЧЕНИЕ ОБЪЕКТЫ И ПРЕДМЕТА

## 1.1 О приложении PolyAbit

Раньше, подавляющее число университетов предоставляло свой сервис для приема заявок от абитуриентов и свою систему их обработок. Эти системы зачастую были не лишены следующих недостатков:

1. Сложный интерфейс для создания заявки
2. Сервис «ложился» от высоких нагрузок

Все это вызывало недовольство среди абитуриентов. Корнем этих проблем является то, что такие сервисы делались уже на устаревших технологиях и методиках, либо давно не обновлялись.

Одной из целей работы является создание прототипа сервиса, лишенного этих проблем. Его название – PolyAbit (сокращенное Polytechnic Abiturient).

Он должен предоставлять мобильное приложение абитуриенты для подачи заявок и панель администратора для членов приемной комиссии.

## 1.2 Историческая справка

История веб-сервисов тесно переплетается с развитием интернета. Первые предпосылки появились в конце 90-х годов, когда возникла потребность в обмене данными между различными системами.

В 1998 году компания IBM представила концепцию "web services", основываясь на использовании XML для формата данных и SOAP для передачи сообщений. Эта концепция получила широкое распространение, заложив фундамент для современной архитектуры веб-сервисов.

Начиная с 2000-х годов, веб-сервисы стали развиваться стремительно, появлялись новые стандарты и технологии. REST API, основанный на использовании HTTP, стал популярным альтернативным подходом к SOAP. Развитие облачных платформ привело к появлению сервисов "Software as a Service" (SaaS), таких как Google Docs и Salesforce.

В 2010-х годах в сфере веб-сервисов произошла революция, связанная с появлением концепции "микросервисов". Микросервисы представляют собой небольшие, независимые приложения, которые выполняют конкретную функцию и могут быть легко интегрированы с другими сервисами. Этот подход позволил разработчикам создавать более гибкие, масштабируемые и устойчивые приложения.

## 1.3 Исследование и обзор технологий

### 1.3.1 Исследование и обзор технологий для создания мобильных приложений

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 [1] – 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. На схеме №1 изображен путь компонента для IOS, на схеме №2 – для Android

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.

### 1.3.2 Исследование и обзор технологий для создания серверных приложений

Язык Go[2] – 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. Вторая, это коммуникация сервисов между собой.

Коммуникации между разделить на синхронные и асинхронные. Первые предполагают получение ответа сразу же, 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.

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 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-сайтами. Для этого нам придется написать прокси сервер (здесь и далее будем называть его API Gateway), который бы транслировал запросы из http в gRPC формат. В случае Go можно воспользоваться плагином grpc-gateway, который сгенерирует прокси-сервер

### 1.3.3 Исследование и обзор технологий для создания веб-сайтов

Веб-сайты, или как чаще всего называют frontend – это та часть приложения, которая непосредственно запускается в браузере.(∧)𝐴∧𝐵𝐴𝐵

Изначально веб-сайт представлял собой множество html файлов, лежащих на сервере. Такой подход называет MPA – multi page application. Постепенно приложения становились сложнее, из-за чего создавать их становилось сложнее. Логичным решением было бы переход к новому подходу – SPA.

SPA – single page application. Данный подход заключается в том, что теперь сервер отдает только один html файл, в котором уже есть скрипты для дальнейшей работы приложения. Это стало возможным благодаря развитию языка JavaScript. В последствие появились инструменты, облегчающие создание SPA приложений. Сначала это был jQuery, затем появился первый полноценный фреймворк Angular, на замену которому пришел React. На данный момент именно React [1] является главным инструментом создания веб-сайтов.

Параллельно этому шло развитие UI-библиотек, предоставляющий разработчикам уже готовый пакет со всеми UI компонентами: кнопки, поля ввода, модальные окна и т.п. Самой популярной библиотекой стал Bootstrap [4].

# ГЛАВА 2. РЕАЛИЗАЦИЯ СОВРЕМЕННОГО ВЕБ-СЕРВИСА

## 2.1 Серверное приложение

**Сначала напишем контракты для взаимодействия, то есть proto файлы.**

|  |  |
| --- | --- |
| 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"  };  }  … // декларация остальных методов  }  **message** **Direction** {  **int64** id = **1**;  **string** code = **2**;  }  … // 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. |

**Таблица №1. Proto-файл с комментариями**

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

**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**

**Рассмотрим этот принцип на архитектуре приложения:**

|  |  |
| --- | --- |
| - cmd  - content  - main.go  - migrator  - main.go  - config  - local.yaml  - go.mod  - go.sum  - internal  - app  - app.go  - grpc  - app.go  - clients  - auth  - grpc  …  - 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 |

Таблица №2. Архитектура сервиса с комментариями

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

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

**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))

}

}()

}

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

… // запускаем http

}

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

… // запускаем http

}

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:

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

GRPCServer \*grpcapp.App

}

**func** New(

log \*slog.Logger,

cfg \*config.Config,

) \*App {

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

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))

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.

## 2.2 Мобильное приложение

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

|  |  |
| --- | --- |
| - 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  - assets  - fonts  - PTSans-Regular.ttf  - images  - …  - babel.config.js  - components  - AnimatedEmoji.tsx  - ScreenLayout.tsx  - …  - constants  - …  - contexts  - auth.tsx  - expo-env.d.ts  - hooks  - useColorScheme.ts  - useStorageState.ts  - …  - 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 |

Таблица №3. Структура мобильного приложения

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);

}

};

**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<{

… // значение по-умолчанию

}>({

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

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

session: **null**,

isLoading: **false**,

});

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

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

**return** (

… // пробрасываем данные в компоненты

}

Контекст – это прослойка, предоставляющая реализациям все необходимые методы для работы с авторизацией пользователя: методы логина и выхода из приложения, а также сам JWT-токен.

Сам хук возвращает переменную и функцию для ее изменения. Само взаимодействие с хранилищем реализовано при помощи библиотеки expo-secure-store. Посмотрим на использование в 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 в приложении есть все необходимое для работы с контентом. Рассмотрим, например, редактирование личной информации:

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

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

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

… // обработка

};

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

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

method: "PUT",

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

});

… // обработка

};

useEffect(() => {

fetchProfile();

}, []);

**return** (

.. // верстка

);

}

Здесь реализован метод для получения профиля и его редактирования.

**ГЛАВА 3. АПРОБАЦИЯ РЕЗУЛЬТАТОВ ИССЛЕДОВАНИЯ**

## 3.1. Введение

Так как работа выполнялась в паре, сайт написана Александр Мураев, группа 5130203/30002.

Апробация проходила путем отправки запросов с клиента, который был представлен ранее, на сервер, для простоты восприятия использовалось приложение Postman[5]

## 3.2. Тестирование запросов

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

Автоматически созданное описание***

Рис.5. Тест POST запроса на авторизацию

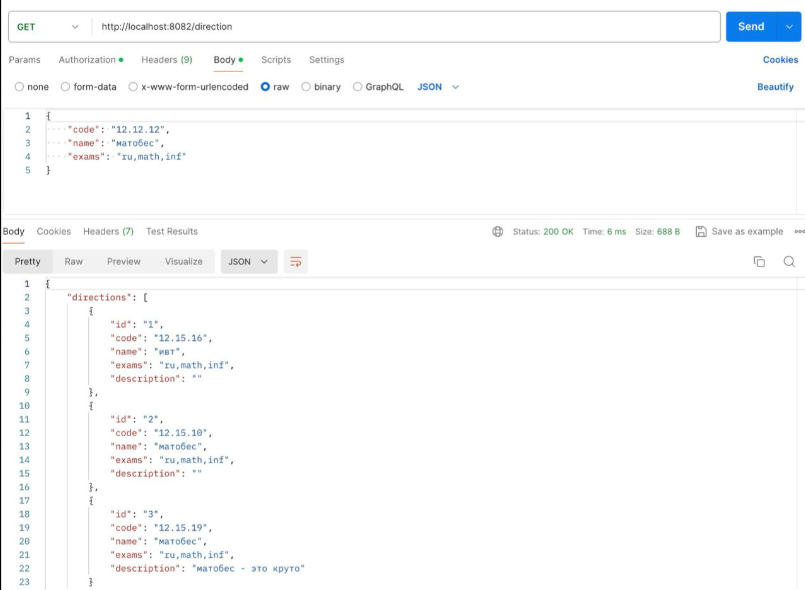
******

Рис.6. Тест GET запроса на получение данных с сервера

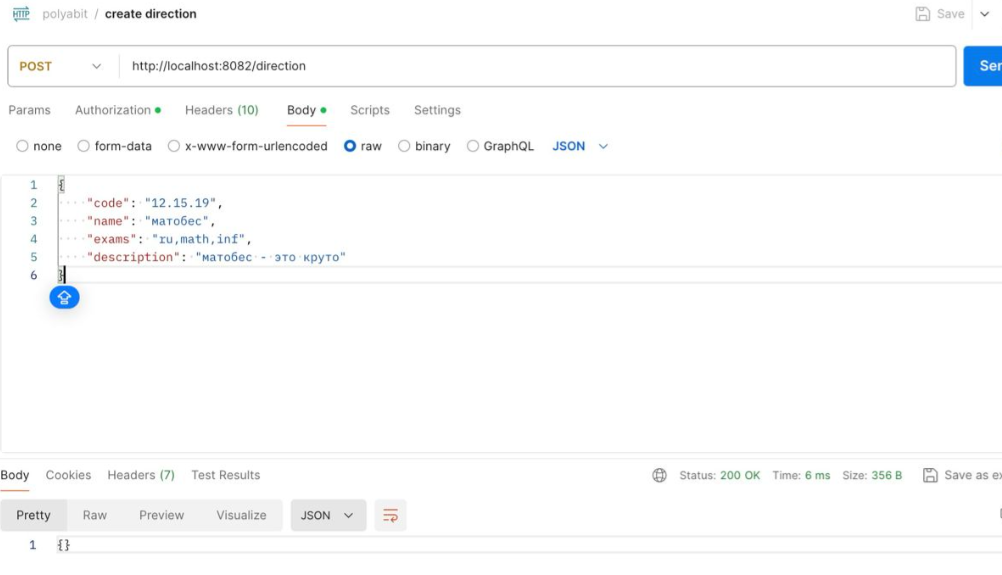
******

Рис.7. Тест POST запроса на отправку данных на сервер

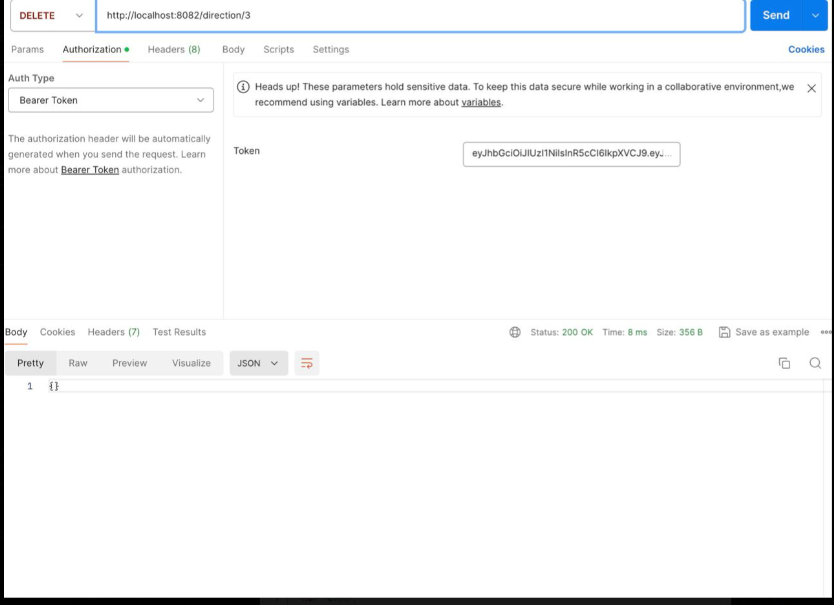


Рис.8. Тест DELETE запроса на удаление данных с сервера

# ЗАКЛЮЧЕНИЕ

Научно-исследовательская работа, посвященная изучению современных веб-сервисов успешно завершена. Все поставленные задачи были решены:

* Проведен глубокий анализ основных архитектур и инструментов для создания веб-сервисов
* Посредством актуальных инструментов создан прототип веб-сервиса.

Таким образом, цель исследования веб-сервисов достигнута. Полные версии программ доступны по ссылке <https://github.com/PolyAbit>

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

*Документация React Native* ─ URL:<https://reactnative.dev/docs/getting-started>

*Документация Golang* ─ URL: <https://go.dev/doc/>

*Документация React* ─ URL: [*https://ru.legacy.reactjs.org/docs/getting-started.html*](https://ru.legacy.reactjs.org/docs/getting-started.html)

*Документация React Boostrap* ─ URL: [*https://react-bootstrap.netlify.app*](https://react-bootstrap.netlify.app)

Инструмент для тестирования стиль – URL: <https://www.postman.com>