МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Национальный исследовательский университет
«МЭИ»

Институт информационных и вычислительных технологий
Факультет прикладной математики и информатики
Кафедра прикладной математики и искусственного интеллекта

Курсовая работа

Сервис регистрации и аутентификации.

Выполнил:
студент 3 курса
группы А-05-19
Каменев Р.Б.

Проверил:
Ивлиев С.А.

Москва 2021

Оглавление

Задача

Используя архитектурный стиль REST, написать сервис регистрации и аутентификации пользователей.

Используемые инструменты

В качестве языка программирования для реализации курсовой работы был выбран язык GO, также у приложения имеется база данных(была выбрана реляционная база данных Postgres, запущенная в docker – контейнере).

Язык GO

Go (часто также golang) — компилируемый многопоточный язык программирования, разработанный внутри компании Google.

Postgres

Postgres - это свободно распространяемая объектно-реляционная система управления базами данных (ORDBMS), наиболее развитая из открытых СУБД в мире и являющаяся реальной альтернативой коммерческим базам данных.

Docker

Docker — это платформа для разработки, доставки и запуска контейнерных приложений. Docker позволяет создавать контейнеры, автоматизировать их запуск и развертывание, управляет жизненным циклом. Он позволяет запускать множество контейнеров на одной хост-машине[1].

Контейнеры — это способ упаковать приложение и все его зависимости в единый образ. Этот образ запускается в изолированной среде, не влияющей на основную операционную систему. Контейнеры позволяют отделить приложение от инфраструктуры: разработчикам не нужно задумываться, в каком окружении будет работать их приложение, будут ли там нужные настройки и зависимости. Они просто создают приложение, упаковывают все зависимости и настройки в единый образ. Затем этот образ можно запускать на других системах, не беспокоясь, что приложение не запустится.

---

[1] **Хост** (от англ. host — «владелец, принимающий гостей») — любое устройство, предоставляющее сервисы формата «клиент-сервер» в режиме сервера по каким-либо интерфейсам и уникально определённое на этих интерфейсах. В более широком смысле под хостом могут понимать любой компьютер, подключённый к локальной или глобальной сети.

Теоретический материал

Архитектурный стиль REST

REST (Representational state transfer) – это стиль архитектуры программного обеспечения для распределенных систем, таких как World Wide Web, который, как правило, используется для построения веб-служб. Термин REST был введен в 2000 году Роем Филдингом, одним из авторов HTTP-протокола. Системы, поддерживающие REST, называются RESTful-системами.

Если говорить коротко и просто, то **REST** – это набор правил (6 правил) о том, как программисту организовать написание кода серверного веб-приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать.

HTTP, URL и Endpoint

RESTful(приложение, в котором все 6 правил учтены) приложение предоставляет список **url-адресов**, с помощью которых сервер может принимать запросы на получение, сохранение, обновление и удаление данных.

Обращение к **url-адресам** происходит с помощью HTTP-методов: GET(получить), POST(сохранить), PUT(обновить), DELETE(удалить). Url может иметь несколько **эндпоинтов**.

Эндпоинт (Endpoint - конечная точка) — это само обращение к маршруту отдельным HTTP методом. Эндпоинт выполняют конкретную задачу, принимают параметры и возвращают данные Клиенту.

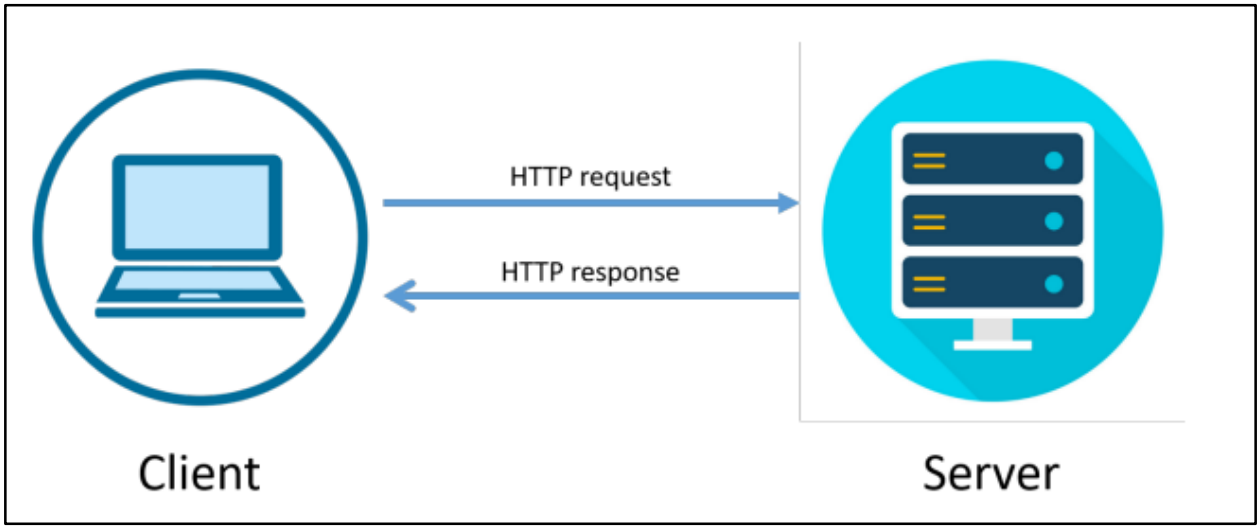


Рисунок 1(схема работы HTTP-запросов)

Конкретное приложение может иметь список url-адресов, с помощью этих адресов мы предоставляем клиенту(или же другим разработчикам) интерфейс, с помощью которого можно взаимодействовать с приложением. Этот интерфейс называется **API**

API

API (Application programming interface) — это контракт, который предоставляет программа(«Ко мне можно обращаться так и так, я обязуюсь делать то и это», если говорить грубо).

API включает в себя:

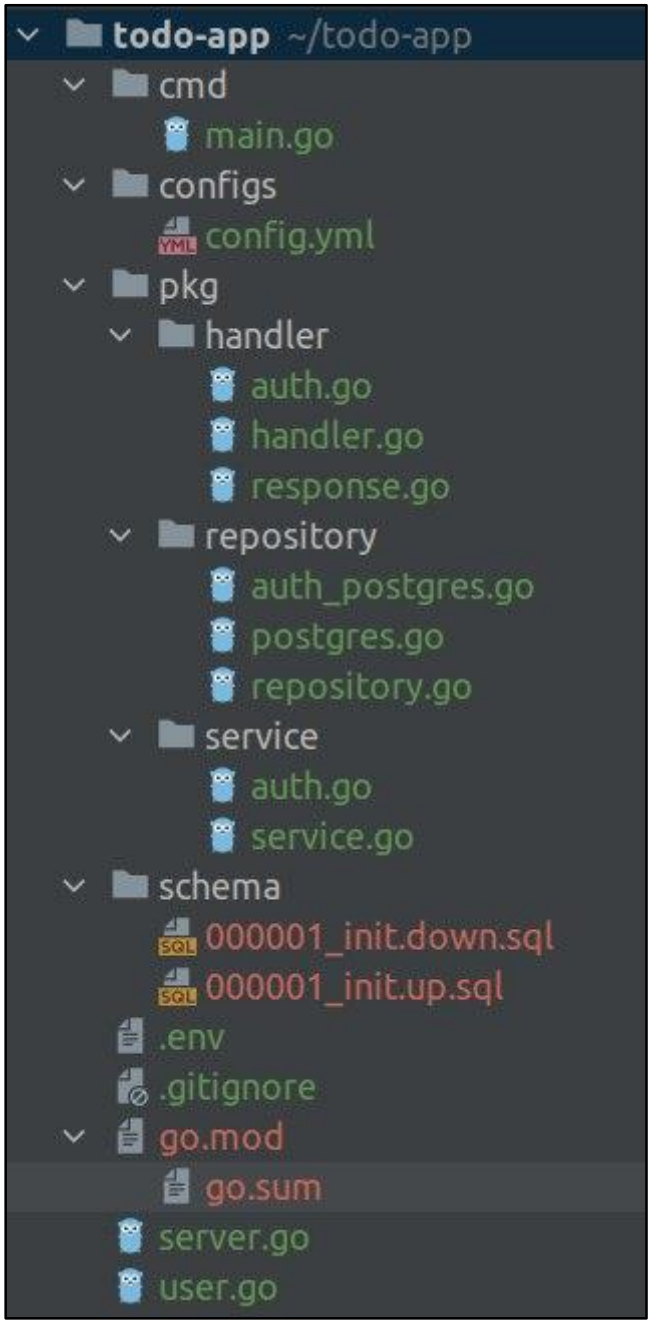
- саму операцию, которую мы можем выполнить,
- данные, которые поступают на вход,
- данные, которые оказываются на выходе (контент данных или сообщение об ошибке).

Такая структура очень схожа с таким понятим как функция(метод) в языке программирования. Да, API - это набор функций. Это может быть одна функция, а может быть много.

Программная реализация

Структура проекта

Будем придерживаться стандартной структуры golang-проектов[2]



[2] Стандартная структура golang-проектов - структура golang-проектов

Описание структуры проекта:

todo-app – файл проекта.

cmd/main.go – точка запуска приложения.

configs/config.yml – файл конфигов(номер порта сервера и данные для подключения к БД).

pkg – здесь расположена основная логика нашего приложения

handler – здесь расположена логика работы с HTTP-запросами, тут мы принимаем запросы от пользователей (на регистрацию и аутентификацию), обрабатываем их и передаём данные на уровень ниже (в **service**).

service – здесь расположены функции создания пользователей и аутентификации пользователей. При регистрации нового пользователя происходит его добавление в БД.

repository – здесь расположена реализация БД (подключение к БД, добавления нового пользователя, получение существующего пользователя по id).

schema – файлы миграции[3] БД

todo-app/server.go – структура сервера и метод для его запуска, остановки.

todo-app/user.go – структура пользователя (поля структуры полностью совпадают с полями в БД).

[3] Миграция БД – переход БД на новую структуру без потери данных (см. более подробнее о миграциях -миграции)

Структура БД

Структура БД очень простая:

users – название БД			
id	name	username	password-hash

id – номер пользователя в БД

name – имя пользователя (подразумевается реальное имя пользователя)

username – имя пользователя (псевдоним)

password-hash – хешированный пароль пользователя.

Механизм регистрации

Код реализации регистрации:

```
func (h *Handler) signUp(c *gin.Context) {
  var input todo.User

  if err := c.BindJSON(&input); err != nil {
    newErrorResponse(c, http.StatusBadRequest,
err.Error())
    return
  }

  id, err :=h.services.Authorization.CreateUser(input)
  if err != nil {
    newErrorResponse(c, http.StatusInternalServerError,
err.Error())
    return
  }

  c.JSON(http.StatusOK, map[string]interface{}{
    "id": id,
  })
}

  c.JSON(http.StatusOK, map[string]interface{}{
    "id": id,
  })
}
```

Сначала мы создаём объект input структуры User. В эту структуру мы будем записывать данные, которые передаёт пользователь при регистрации.

Структура User:

```
type User struct {
  Id       int    `json:"-" db:"id"`
  Name     string `json:"name" binding:"required"`
  Username string `json:"username" binding:"required"`
  Password string `json:"password" binding:"required"`
}
```

Поля этой структуры полностью совпадают с полями БД (ведь именно эти данные нужны для регистрации).

У каждого поля мы определили json-теги (это нужно для того, чтобы тело запроса HTTP, которое приходит к нам в json, автоматически распарсилось по полям данной структуры. Например, данные в запросе с полем “name” будут записаны в поле Name структуры User и т.д. для остальных полей). Также, мы прописали теги “binding: required”, этот тег проверяет наличие поля в запросе. Например, если пользователь не укажет Username при регистрации, то ему вернётся ошибка.

Вернёмся к коду регистрации, метод BindJson валидирует поля запроса, если произошла ошибка (было упомянуто выше, в каких случаях может произойти ошибка), то вызываем ошибку `http.StatusBadRequest` (код ошибки 400) – этот код означает, что пользователь передал некорректные данные для регистрации. Если ошибки нет, то вызываем метод CreateUser, в которой передаём данные для регистрации (успешно сформированную структуру User). Внутри себя функция CreateUser хеширует пароль (записывает вместо пароля хешированный пароль в поле password структуры User) и уже с обновлённой структурой User вызывает метод CreateUser у БД.

CreateUser у БД делает INSERT-запрос (запрос на создание новой записи в БД) и возвращает id записи в БД (если произошла ошибка при запись в БД, возвращаем id, равный 0 и ошибку).

Далее, если функция CreateUser вернула ошибку, то вызываем ошибку `http.StatusInternalServerError` (код ошибки 500) – этот код означает, что произошла какая-то внутренняя ошибка на сервере.

Если ошибки не произошло, возвращаем сгенерированный id в json -формате и `http.StatusOK` (код 200) – этот код означает, что запрос успешен.

Опишем механизм вызова ошибок. Код вышеупомянутой функции newErrorResponse:

```
type error struct {
  Message string `json:"message"`
}

func newErrorResponse(c *gin.Context, statusCode
int, message string) {
  logrus.Error(message)
  c.AbortWithStatusJSON(statusCode,
error{message})
}
```

Создаём структуру с названием error (именно такого типа будут ошибки, связанные с работой сервера). Эта структура будет иметь всего лишь одно поле – Message, типа string (строка с ошибкой).

Создадим функцию newErrorResponse, которая будет принимать контекст[4] (если точнее, то контекст из фреймворка gin. С помощью контекста в go можно прерывать выполнение запроса, выполняемого в го-рутине[5] (потоке), передавать специфичные параметры для запроса), код ошибки и строку ошибки. Сначала записываем ошибку в лог, а затем у контекста вызываем метод AbortWithStatusJSON, передав ему код-ошибки и сконструированную структуру для ошибки (2-ым параметром AbortWithStatusJSON принимает интерфейс, поэтому передача структуры error допустима). Поскольку у одного эндпоинта может быть несколько последовательных обработчиков, то AbortWithStatusJSON блокирует выполнение последующих обработчиков, а также записывает в ответ статус-код и тело сообщения в формате json.

[4] Статья о контекстах - контексты
[5] Статья о го-рутинах – го-рутины

Механизм аутентификации

Аутентификация - это процесс проверки учётных данных пользователя (логин/пароль). Проверка подлинности пользователя путём сравнения введённого им логина/пароля с данными сохранёнными в базе данных.

В моём проекте был использован механизм аутентификации с помощью JWT-токенов[6].

Конечно, для такого относительно лёгкого процесса как аутентификация можно было и не использовать токены, но в будущем планируется реализовать механизм авторизации[7]. При авторизации необходимо проверять, имеет-ли право данный пользователь выполнять те или иные действия на сайте, что относительно просто можно сделать через токены.

Процесс аутентификации с помощью JWT-токенов делится на 2 части:

- проверка наличия пользователя в БД с таким логином и паролем
- если пользователь с таким логином и паролем есть в БД, то сгенерировать для него токен (как раз этот токен мы будем использовать в дальнейшем при авторизации).

[6] См. более подробно про механизм аутентификации с помощью JWT-токенов здесь
[7] Авторизация(authorization — разрешение, уполномочивание) - это проверка прав пользователя на доступ к определенным ресурсам(см. более подробно здесь).

Аутентификация

Код реализации:

```
func (h *Handler) signIn(c *gin.Context) {
  var input signInput

  if err := c.BindJSON(&input); err != nil {
    newErrorResponse(c, http.StatusBadRequest,
err.Error())
    return
  }

  token, err :=
h.services.Authorization.GenerateToken(input.Username,
input.Password)
  if err != nil {
    newErrorResponse(c, http.StatusInternalServerError,
err.Error())
    return
  }

  c.JSON(http.StatusOK, map[string]interface{}{
    "token": token,
  })
}
```

Для аутентификации нам нужно имя пользователя (username) и пароль, поэтому структура User нам уже не подойдёт (т.к. у этой структуры обязательно наличия поля имени пользователя (Name)).

Для этой цели создадим новую структуру signInput:

```
type signInput struct {
  Username string `json:"username" binding:"required"`
  Password string `json:"password" binding:"required"`
}
```

У этой структуры, как и у прошлой, также есть json-теги и теги “binding:"required"”.

Код реализации метода signIn очень схож с кодом реализации signUp. Единственное отличие в том, что вместо метода CreateUser мы вызываем GenerateToken, рассмотрим этот метод подробнее:

```
type tokenClaims struct {
  jwt.StandardClaims
  UserId int `json:"user_id"`
}

func (s *AuthService) GenerateToken(username, password string) (string,
error) {
  user, err := s.repo.GetUser(username, generatePasswordHash(password))
  if err != nil {
    return "", err
  }

  token := jwt.NewWithClaims(jwt.SigningMethodHS256, &tokenClaims{
    jwt.StandardClaims{
      ExpiresAt: time.Now().Add(tokenTTL).Unix(),
      IssuedAt:  time.Now().Unix(),
    },
    user.Id,
  })

  return token.SignedString([]byte(signedKey))
}
```

С помощью метода GetUser репозитория мы обращаемся к БД для проверки наличия пользователя в БД по переданному логину и паролю (внутри функции GetUser делаем SELECT-запрос к нашей БД, в запросе передаём имя пользователя и предварительно хешированый пароль (ведь в БД лежат хешированные пароли)). Если такого пользователя нет, то возвращаем пустой токен и ошибку от БД.

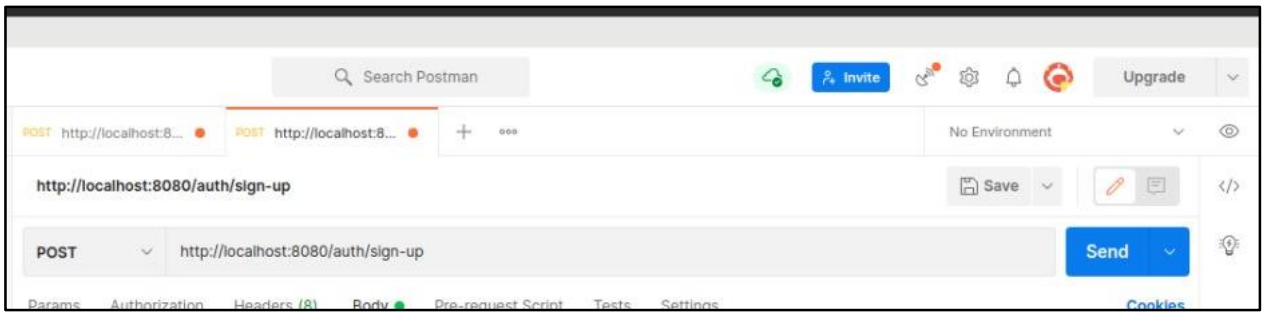
Если ошибки нет, то вызываем метод NewWithClaims (этот метод нужен для создания экземпляра, с помощью которого и будет создаваться токен, с помощью данного метода мы зададим “настройки” нашего будущего токена, а именно: метод подписи (HS256) и стандартные требования к токену (например, время жизни токена). Список требования к токену можно переопределять, сделаем это, добавив к требованиям id пользователя. Для этого создадим структуру tokenClaims, в которой будет 2 поля: id пользователя и стандартные требования к токену (этих стандартных требований достаточно много, мы не обязаны их определять все). Непосредственно в коде мы в стандартных требованиях к токену мы задаём его время жизни (12 часов) и время создания.

Из функции вернём созданный токен, создаём мы его с помощью метода SignedString, в этот метод передаём так называемый “секретный ключ” (с помощью этого ключа мы будем расшифровывать токен в дальнейшем).

Тесты регистрации и аутентификации

Все тесты будем проводить через приложение Postman[8]. Это достаточно мощное приложение, которое позволяем по-разному(и вручную, и через созданные скрипты и иными способами) протестировать созданное API. Ограничимся ручным тестированием

В строке запроса пропишем путь к тестируемому API (в данном случае мы обращаемся к методам группы auth к методу sign-up), тип запроса - POST:

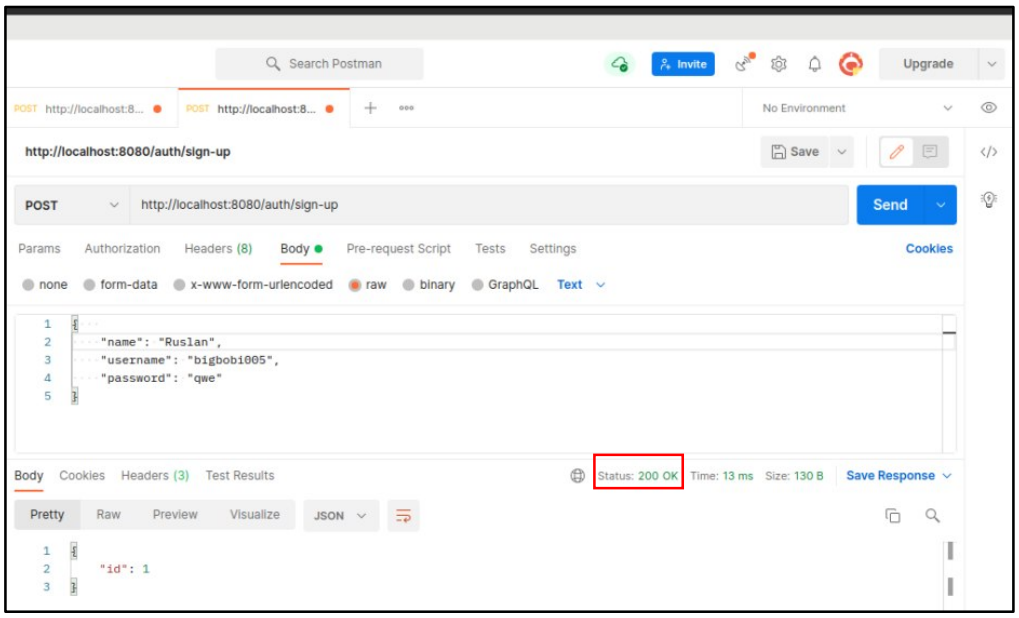


Регистрация

Тест №1

Попробуем зарегистрировать нового пользователя.

Имя пользователя(Name): Ruslan
Имя пользователя (Username): bigbobi005
Пароль: qwe



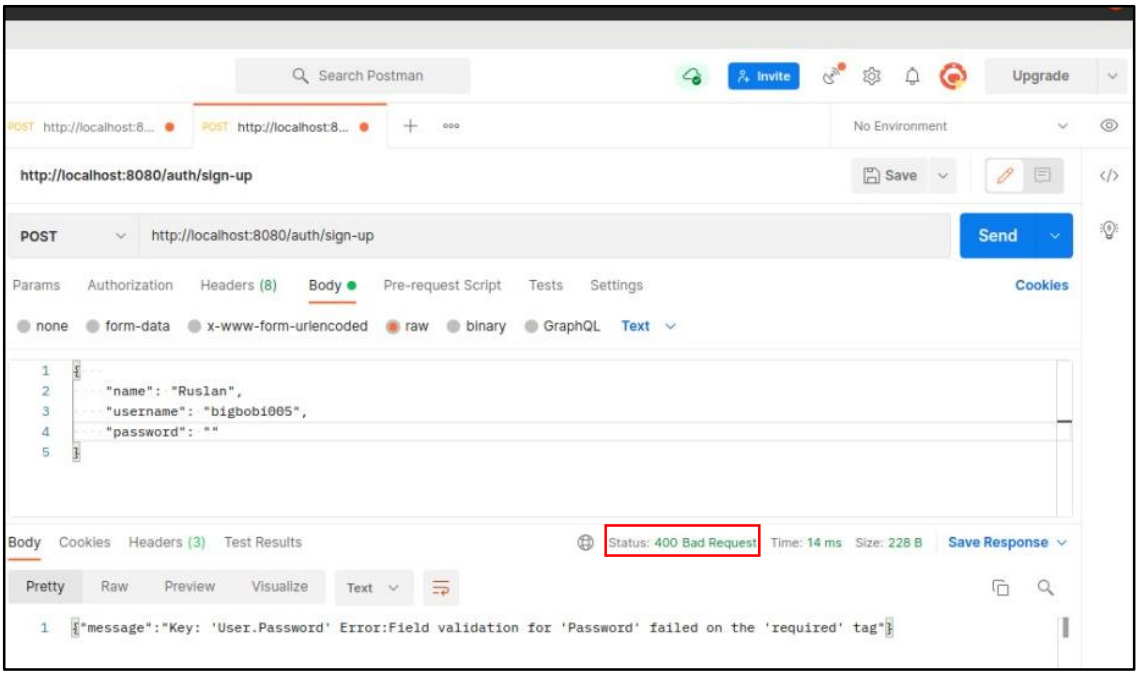
[8] Postman – платформа для тестирования API ()

Новый пользователь успешно создан и добавлен в БД, в качестве ответа получаем id пользователя. Обратим внимания на статус-код ответа – 200, что ещё раз подтверждает, что запрос прошёл успешно.

Тест №2

Попробуем сделать ошибку в запросе, забыв написать в теле запроса пароль.

Имя пользователя(Name): Ruslan
Имя пользователя (Username): bigbobi005
Пароль:



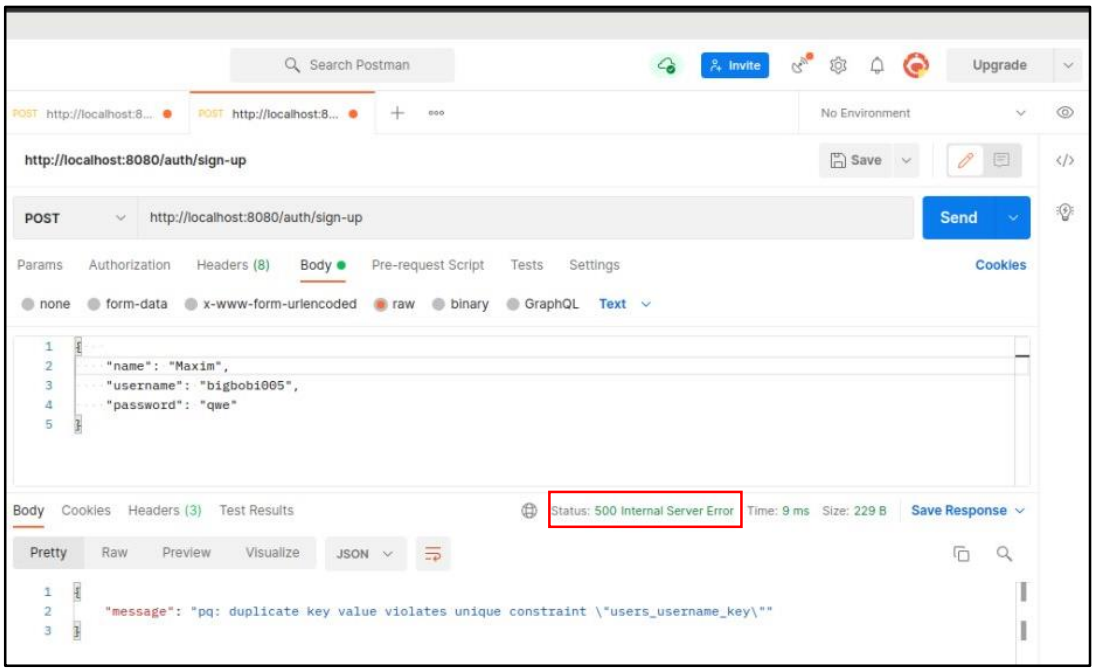
Как мы видим, в качестве ответа вернулась ошибка об отсутствии пароля в запросе. Статус-код ответа – 400 (Пользователя ввёл некорректные данные)

Обычному пользователю, конечно, будет непонятно, если он увидит такую строку ошибки. Работа над пользовательскими интерфейсами, в том числе отправка ответов пользователю – следующая ступень развития проекта.

Тест №2

Попробуем зарегистрировать другого пользователя

Имя пользователя(Name): Maxim
Имя пользователя (Username): bigbobi005
Пароль: qwe



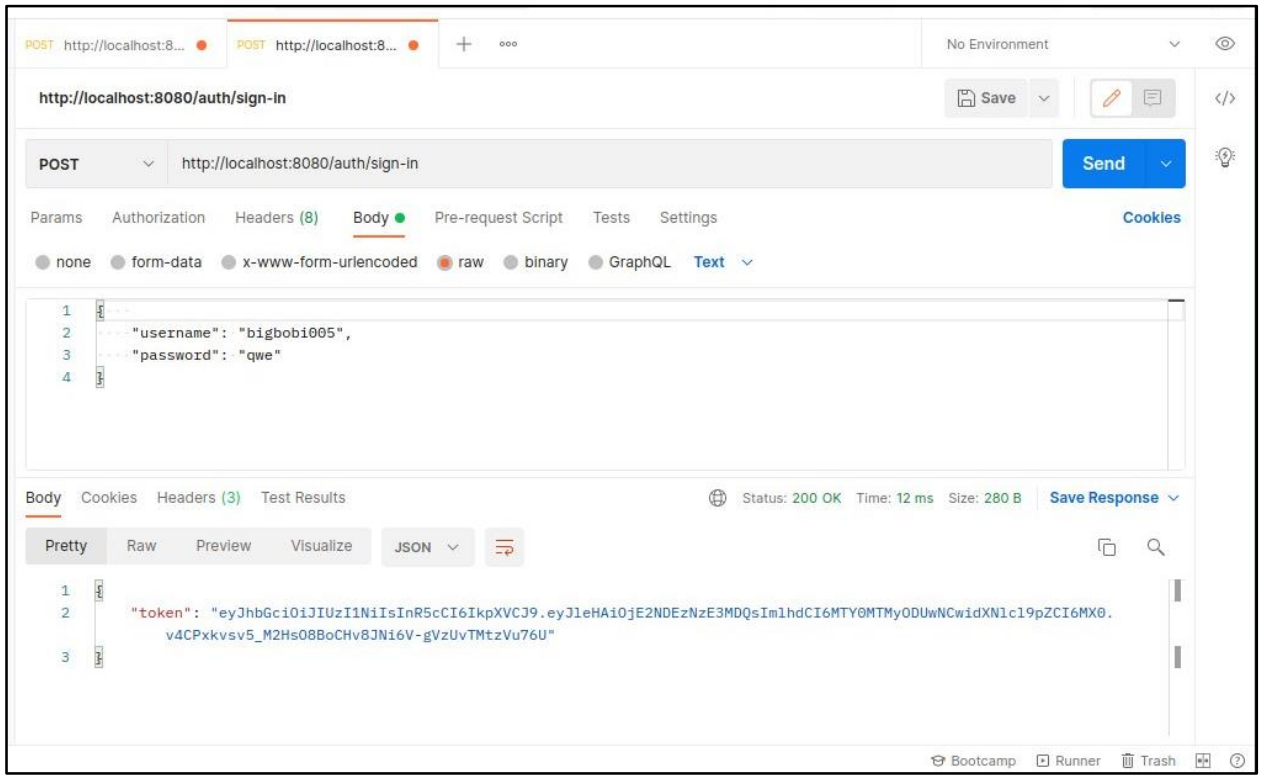
Мы видим ошибку от БД. Статус-код ошибки – 500 (ошибка на сервере)
Ошибка в том, что пользователь с username bigbobi005 уже существует. Я специально задал поле username в БД уникальным.

Аутентификация (Часть 1):

Тест №1

Попробуем аутентифицировать созданного ранне пользователя.

Имя пользователя (username): bigbobi005
Пароль: qwe

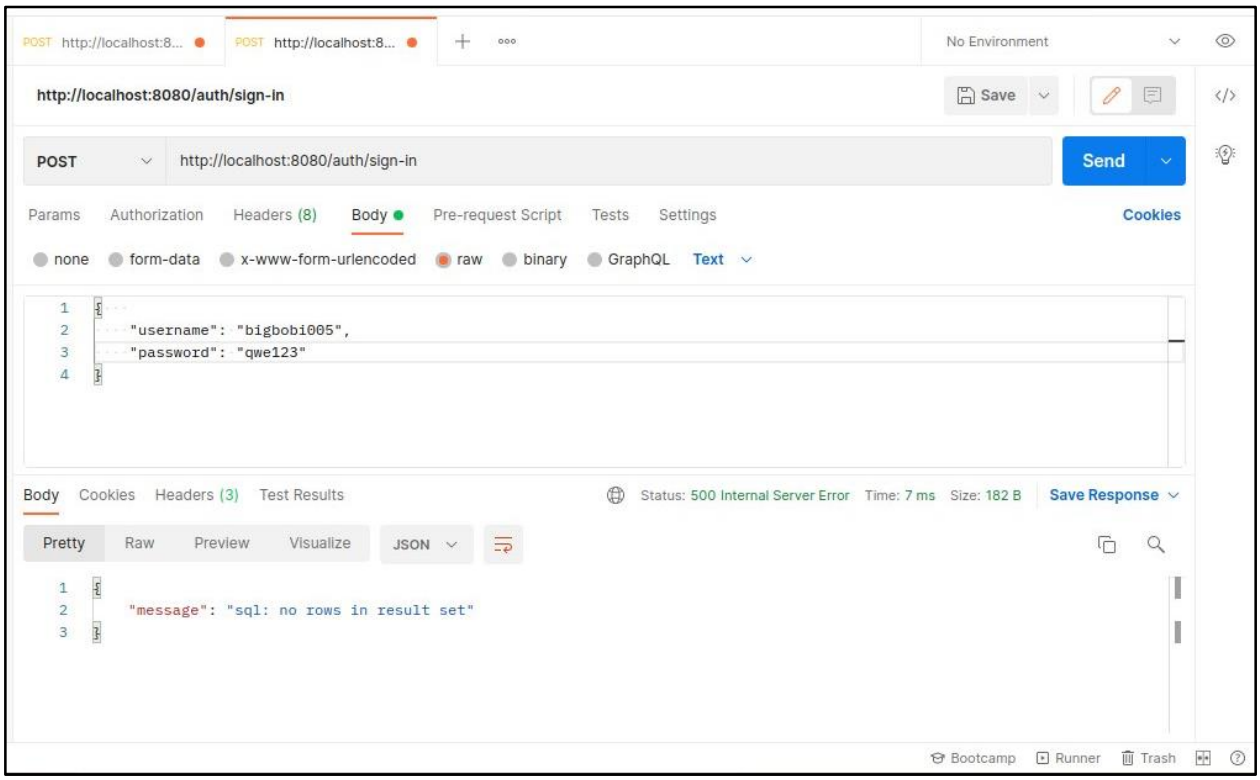


Пользователь успешно аутентифицирован, в качестве ответа мы получили токен для пользователя (в следующей части тестирования аутентификации мы будем сравнивать этот токен с токеном, который имеет пользователь при обращении к какому-нибудь API).

Тест №2:

Попробуем ввести неверный пароль:

Имя пользователя (username): bigbobi005
Пароль: qwe123

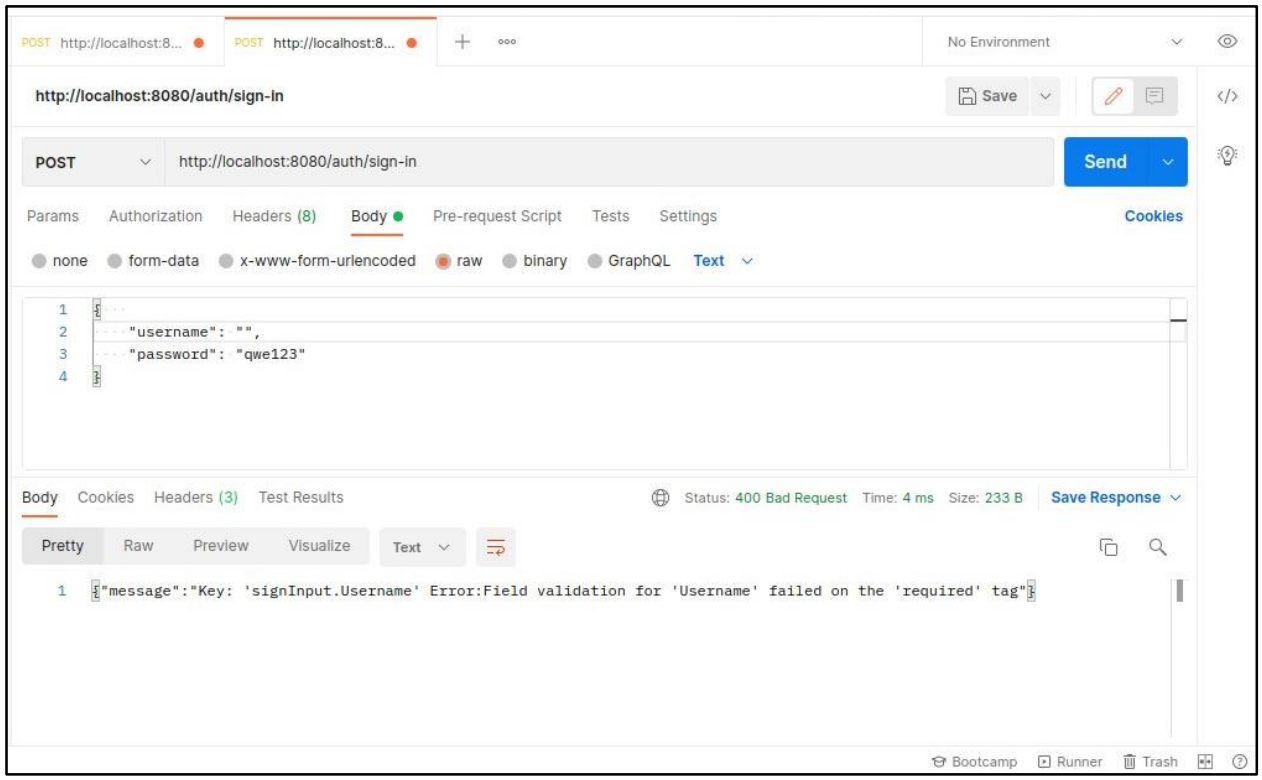


Как мы видим, вернулась ошибка от БД. Ошибка в том, что в БД не записи с таким паролем.

Тест №3

Попробуем оставить поле username пустым:

Имя пользователя (username):
Пароль: qwe



Как мы видим произошла ошибка пользовательского ввода (мы не указали поле Name).

Список литературы и ресурсов:

[2] Структура golang-проектов [Электронный ресурс]: https://github.com/golang-standards/project-layout/blob/master/README_ru.md

[3] Миграция БД [электронный ресурс]: https://webformyself.com/phinx-sistema-migracii-bazy-dannyx/

[4] Контексты [электронный ресурс]: https://habr.com/ru/company/nixys/blog/461723/

[5] Го-рутины [электронный ресурс]: https://golangify.com/goroutines

[6] JWT-токены [электронный ресурс]: https://habr.com/ru/post/340146/

[7] Авторизация и другое [электронный ресурс]: https://gist.github.com/zmts/802dc9c3510d79fd40f9dc38a12bccfc

[8] Postman – [электронный ресурс]: https://www.postman.com/product/what-is-postman/