mtproject-ru

Встреча 2. База данных. Авторизация. DevOps

Дата встречи: 19.02.2026
Формат: оффлайн

1. Пользователи
2. Авторизация
3. События
4. Дни
5. Комнаты

На данных слайдах будет представлен, в основном, функционал БД (модели и требования к репозиториям).
Но для бэкендера указания будут те же самые:

1. Для каждой модели должны быть написаны необходимые DTO
2. Для каждой фичи должен быть сервис
3. Для каждой функции БД (репозитория) должен быть эндпоинт

База данных. Пользователи

```sql
CREATE TABLE IF NOT EXISTS users(
    id BIGINT GENERATED BY DEFAULT AS IDENTITY (
        START WITH 100000
        MINVALUE 100000
        INCREMENT BY 1
        ) PRIMARY KEY,

    username VARCHAR(64) UNIQUE NOT NULL,
    first_name VARCHAR(64) NOT NULL,
    last_name VARCHAR(64),
    password VARCHAR(64) NOT NULL
);

CREATE INDEX IF NOT EXISTS users_username_idx ON users(username);
```

```go
package entities

type UserEntity struct {
    Id       uint   `db:"id"`
    Username string `db:"username"`
    Password string `db:"password"`
}
```

На поле Username создается индекс*

Что там с users(id), много слов?

BIGINT GENERATED BY DEFAULT AS IDENTITY = SERIAL AUTO INCREMENT
Это новый стандарт задания первичных ключей в постгрес. И он позволяет начинать не с 1, а с любого значения (START WITH, MINVALUE, MAXVALUE)

*Индексы в PostgreSQL — специальные объекты базы данных, предназначенные в основном для ускорения доступа к данным. Это вспомогательные структуры: любой индекс можно удалить и восстановить заново по информации в таблице.

Репозиторий должен иметь следующий функционал:

1. Создание пользователя
2. Получение пользователя по id
3. Получение пользователя по username
4. Проверка существования пользователя (по id или username)
5. Обновление пользователя
6. Удаление пользователя

Запрос пользователя
username,
first_name,
last_name или null,
password (не хэширован)

Ответ пользователя
id,
username,
first_name,
last_name или null

Эти структуры используются для всех запросов/ответов

Запрос на проверку существования
id или username

Тут дела обстоят интреснее... Мы будем использовать JWT-аутентификацию с двумя токенами. Вот немного теории:

После создания аккаунта пользователь получает пару токенов: access и refresh токены (1). Refresh токен сохраняется в хранилище *(далее),* а access возвращается в ответе на аутентификацию.

Когда пользователь хочет получить доступ к защищенному ресурсу, то он отправляет этот токен в ЗАГОЛОВКЕ (не куках или тем более теле) «Authorization: Bearer токен». Сервер проверяет, жив ли и валиден access токен, забирает из него user(id) и если пользователь существует, то возвращает нужные данные.
Если же access токен умер, то тут используется уже refresh-токен, с помощью которого генерируется новый access токен и возвращается пользователю вместе с нужными данными.
Ну а если и rehresh умер, то тут 401 (Unauthorized) и пользователю надо будет заново входить в аккаунт.
Цикл повторяется (1).

Ну вот и refresh токен мы будет хранить в REDIS*, потому что он такое любит.

Ключем в нашем случае будет выступать айди пользователя user(id), а значением - refresh токен.
Когда мы достаем из access токена айди пользователя и нам требуется обновить access токен, то мы обращаемся в редис за refresh токеном.

```
type UserToken struct {
    Id uint
    Token string
}
```

Разобраться с подключением и работой с редисом это будет домашним заданием, удачи)
Я проверю потом, чтоб все красивенько было

Рекомендуемая (не обязательно) библиотека редиса: go-redis (ссылка)

*Редис - это высокопроизводительная NoSQL система управления базами данных (СУБД), работающая в оперативной памяти (in-memory) по принципу «ключ-значение». Используется как сверхбыстрый кэш, брокер сообщений и хранилище данных (сессий, профилей, структур) для ускорения работы приложений.

Для работы с токенами и авторизацией вы сами можете выбрать формат,
если он, конечно же, удовлетворяет архитектуре

В бд для токенов должны быть реализованы следующие функции:
1. Внесение пары в БД
2. Получение значения по ключу users(id)
3. Обновление токена по users(id)
4. Удалени токена

Для авторизации необходим следующий функционал:
1. Регистрация пользователя (создание пользователя и токена)
2. Авторизация пользователя (проверка валидности токена и пароля, существования пользователя)
3. Обновление токена (пары)

Как же будем проверять авторизацию?
На каждом приватном ендпоинте должен работать middleware. Проверка
доступа будет осуществляться на бэке!!!
Для этого надо будет написать функцию проверки - тут как хотите

```go
type Mode uint
const (
    MODE_PUBLIC = iota
    MODE_PROTECTED
    MODE_PRIVATE
)

type EventEntity struct {
    Id uint
    OpenMode Mode
    EventTitle string // VARCHAR(128)
    EventDescription sql.NullString // VARCHAR(512)
    EventStart sql.NullTime
    EventEnd sql.NullTime
    EventCreated time.Time
}
```

Репозиторий содержит следующий функционал:

1. Получение события по трем полям:
id, event_created, event_start
2. Обновление события
3. Удаление события
4. Создание события в двух вариантах:
с привязкой ко дню, без привязки ко дню

!Mode будет встречаться в дальнейших структурах - это общий enum, так что надо будет его создать в БД!

Домашнее задание для БДшника:
Написать миграцию для создания таблицы.
Самостоятельно выберите формат для EventCreated
(TIMESTAMP или TIMESTAMPZ)

```
type Mode uint
const (
    MODE_PUBLIC = iota
    MODE_PROTECTED
    MODE_PRIVATE
)
```

Вообще, мне кажется, что стоит переименовать Mode в VISIBILITY - на ваше усмотрение!

MODE_PUBLIC = доступно для всех пользователей любой может зайти и посмотреть

MODE_PROTECETD = доступно для ограниченного круга лиц

MODE_PRIVATE = ни для кого не доступно, только для самого пользователя

Как же указывать круг лиц, для которых виден объект?
Тут у меня нет единого решения, два варианта:
1. Добавить к каждому объекту поле VisibleFor []uint - массив айди объектов, для которых виден данный объект
2. Для каждого объекта создать отдельную реляцию и там плясать

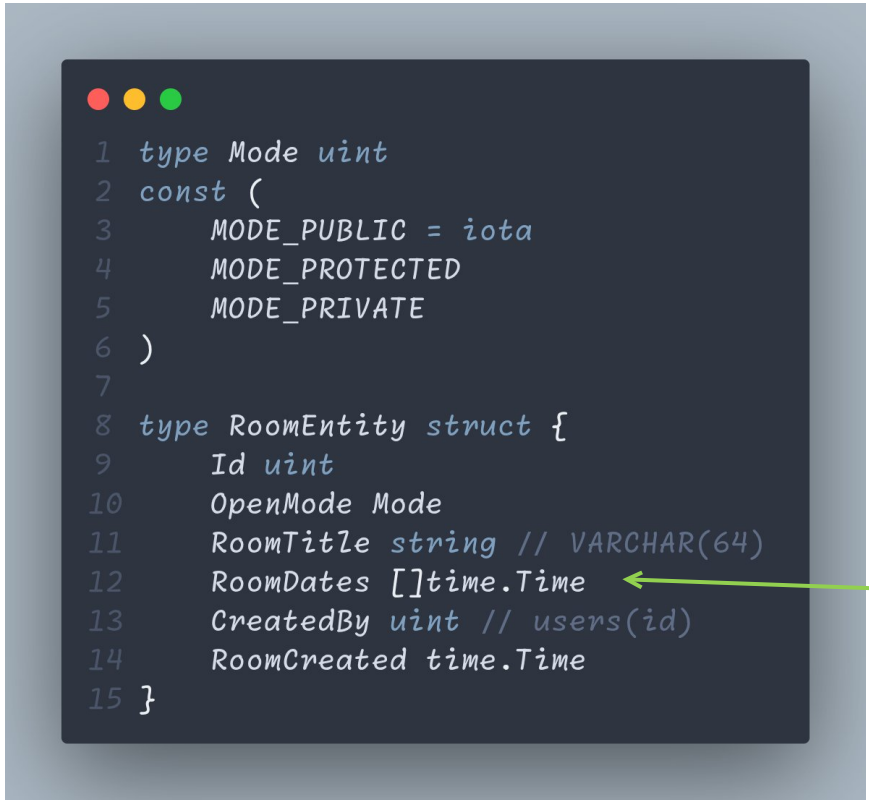
Почему второй вариант вообще существует? Потому что я не уверен по поводу производительности и удобства массивов. Так что этот вопрос вы решаете как хотите сами!

```go
type Mode uint
const (
    MODE_PUBLIC = iota
    MODE_PROTECTED
    MODE_PRIVATE
)

type DayEntity struct {
    Id uint
    OpenMode Mode
    Date time.Time
}

```

Репозиторий содержит следующий функционал:

1. Создание дня
2. Изменение режима видимости OpenMode
3. Получение дня по id, date

Домашнее задание для БДшника:
Написать миграцию для создания таблицы.
Самостоятельно выберите формат для Date
(нам тут не нужно время, а только
год месяц день - какой формат подойдет?)



```go
type Mode uint
const (
    MODE_PUBLIC = iota
    MODE_PROTECTED
    MODE_PRIVATE
)

type RoomEntity struct {
    Id uint
    OpenMode Mode
    RoomTitle string // VARCHAR(64)
    RoomDates []time.Time
    CreatedBy uint // users(id)
    RoomCreated time.Time
}
```

Репозиторий содержит следующий функционал:

1. Создание комнаты
2. Добавления дня(даты) в RoomDates
3. Обновление названия комнаты
4. Изменение видимости OpenMode
5. Удаление комнаты
6. Получение все дней(RoomDates) по id комнаты
7. Получение всех публичных комнат (MODE_PUBLIC)?

Про RoomDates:
Это массив* дат, которые используются в данной комнате.
У них такой же тип, как и у days(date).

Домашнее задание для БДшника:
Написать миграцию для создания таблицы.
Самостоятельно выберите формат для
RoomCreated (TIMESTAMP или TIMESTAMPZ)

*В postgresql есть массивы, а как с ними работать?
Вот информация на эту тему (ссылка)

```go
type UserToDay struct {
    UserId uint // users(id)
    DayId uint // days(id)
}

type UserToEvent struct {
    UserId uint // users(id)
    EventId uint // events(id)
}

type DayToEvent struct {
    DayId uint // days(id)
    EventId uint // events(id)
}

// RoomMembers
type RoomToUser struct {
    RoomId uint // rooms(id)
    UserId uint // user(id)
}
```

Дополнения к репозиториям:
Пользователь: привязка дня к пользователю, получение всех дней для пользователя

Событие: -

День: привязка события ко дню, получение всех событий для дня

Комната: добавление пользователя в комнату, получение всех пользователей для комнаты

Задания со звездочкой:
получение количества событий у пользователя,
получение количества события для комнаты,
получение всех событий для комнаты
(тут нужно учитывать Mode событий!) - **задание с двойной звездочкой**

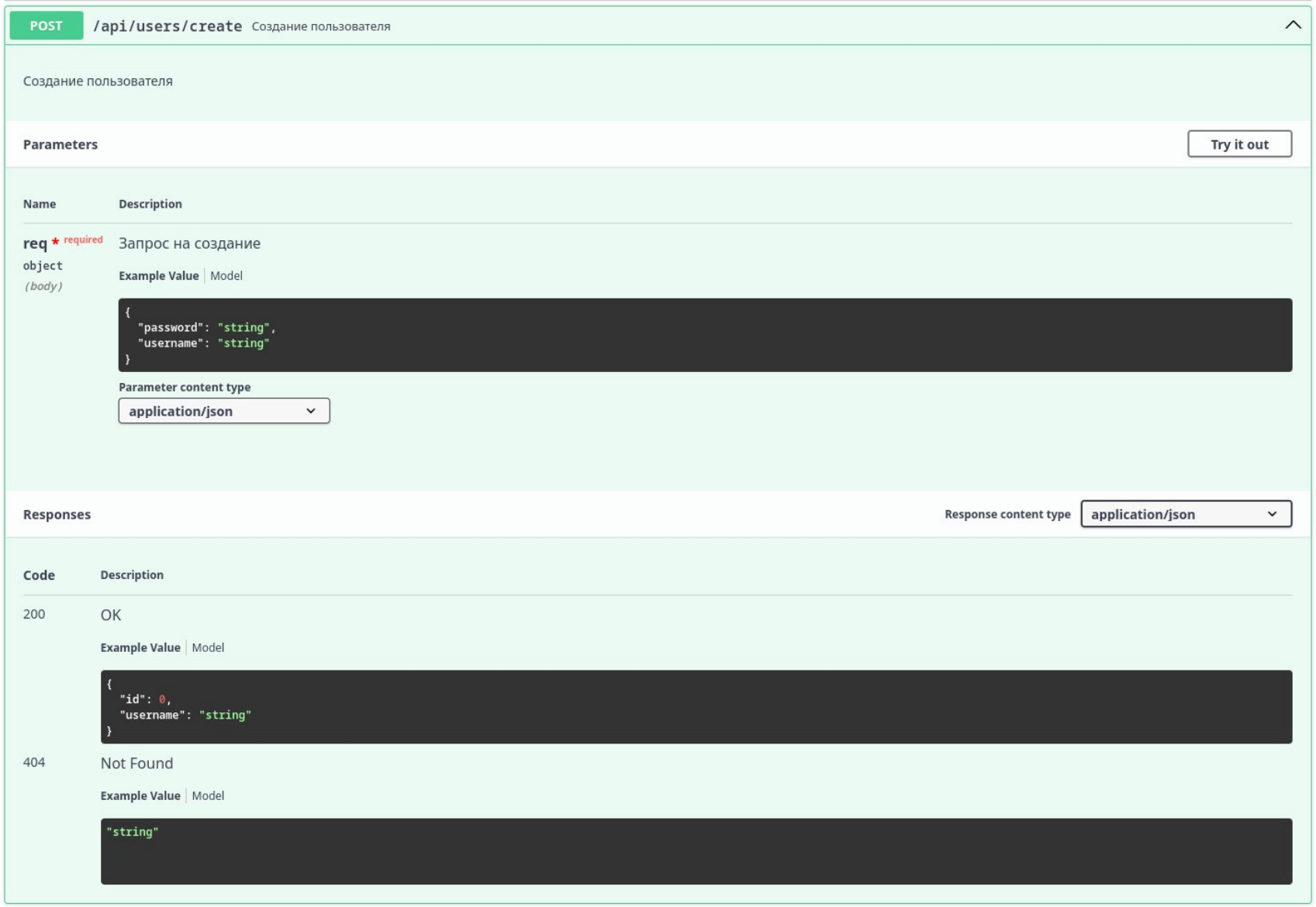
1. БД: пишется модель и интерфейс для репозитория | Бэк: пишутся DTO и mapper-ы
2. БД: пишутся миграции | Бэк: создается сервис, можно реализовывать его
3. БД: пишется имплементация (и реализация) репо | Бэк:
4. БД: тестирование функционала
5: БД: | Бэк: тестируется сервис
6: БД: багфиксы с п.5 | Бэк: пишутся контроллеры
7: БД: багфиксы | Бэк: тестирование контроллеров

Работа над фичей завершена

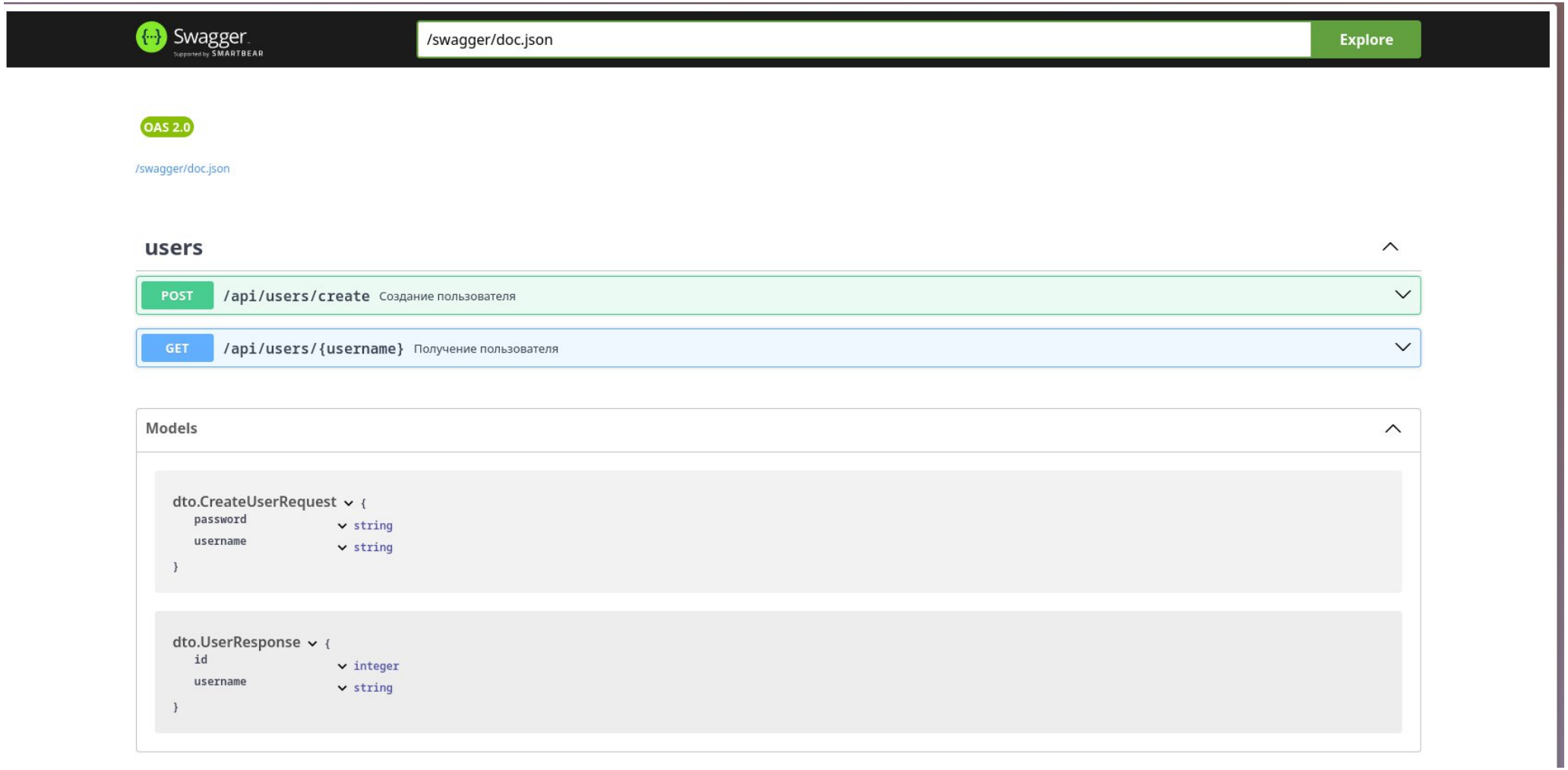
По поводу тестов:
для бд тесты реализуются с помощью стандартных инструментов го

для бэка тесты сервиса с помощью го, для контроллеров с помощью постмана (в таком случае коллекции запросов сохраняются в папке специальной), либо с помощью .http файлов (ссылка) (по туториалам скудно, лучше смотреть страницу расширения в вскоде (ссылка)

Документирование



Документирование



```
// @Summary Создание пользователя
// @Tags users
// @Description Создание пользователя
// @ID create-user
// @Accept json
// @Produce json
// @Param req body dto.CreateUserRequest true "Запрос на создание"
// @Success 200 {object} dto.UserResponse
// @Failure 404 {string} not found!
// @Router /api/users/create [post]
func createUser(ctx fiber.Ctx) error {…
}

// @Summary Получение пользователя
// @Tags users
// @Description Получение пользователя по юзернейму
// @ID get-user-by-username
// @Produce json
// @Param username path string true "Имя пользователя"
// @Success 200 {object} dto.UserResponse
// @Failure 404 {string} not found!
// @Router /api/users/{username} [get]
func getUserByUsernameHandler(ctx fiber.Ctx) error {…
}
```

Мы используем стандарт для REST документации - OpenAPI (ранее Swagger).

Вообще, OpenAPI - это лишь файлы json или yaml, содержащие в себе определенную структуру, но их мы генерировать ручками не будем - это мрак! Этот, казалось бы, небольшой пример занял 70 строк в выходе!

Все это настраивается с помощью надстройки для fiber - swaggo и его cli - swag, который создает в корне папку docs со всем необходимым. А swaggo делает сайт.

В репозитории вы все увидите, я покажу на коде.

Нам нужно будет лишь писать такие вот большие комментации по определенному шаблону (ссылка)

На данном примере представлено документирование только хэндлеров. Как вы могли заметить, в примере у Models не было документации - ее тоже можно сделать! Это вам на домашнее задание

Для девопса мы будем использовать *две** технологии:
Docker и docker compose

У сервера будет два режима (профиля) работы: **production** и **dev**
И там, и там будет использоваться постгрес и редис

Вот только в dev режиме, для облегчения дебага и наглядности данных в обеих БД будут использоваться pgadmin и redis insights соответственно -
это инструменты, которые в реальном времени позволяют графически просматривать базу данных (GUI)

А вот по поводу Dockerfile что?
Догадайтесь, какое это задание...

```yaml
services:
  postgres:
    image: "postgres:18-alpine"
    container_name: "postgres"
    profiles:
      - dev
      - production
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgres/data
    networks:
      - appnet

  pgadmin:
    image: dpage/pgadmin4:latest
    container_name: "pgadmin"
    profiles:
      - dev
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: admin
    ports:
      - "8080:80"
    volumes:
      - pgadmin_data:/var/lib/pgadmin
    depends_on:
      - postgres
    networks:
      - appnet

  redis:
    image: "redis:8-alpine"
    container_name: "redis"
    profiles:
      - dev
      - production
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/var/lib/redis
    networks:
      - appnet

  redis-insight:
    image: redis/redisinsight:latest
    container_name: "redis-insight"
    profiles:
      - dev
    ports:
      - "5540:5540"
    volumes:
      - redis_insight_data:/var/lib/redis_insight
    depends_on:
      - redis
    networks:
      - appnet

volumes:
  pgadmin_data:
  postgres_data:
  redis_data:
  redis_insight_data:


networks:
  appnet:
    driver: bridge
```

*Dockerfile — это скрипт-инструкция для сборки одного образа контейнера (основа).
Docker Compose — это файл YAML, который управляет запуском и взаимодействием нескольких контейнеров как единого приложения (сеть, тома, зависимости).

go-redis - go-redis is the official Redis client library for the Go programming language
https://github.com/redis/go-redis

swag - Swag converts Go annotations to Swagger Documentation 2.0.
https://github.com/swaggo/swag

go-fiber/swagger - This project demonstrates how to integrate Swagger for API documentation in a Go application.
https://docs.gofiber.io/recipes/swagger/

rest-client(.http) - Example
https://kenslearningcurve.com/tutorials/test-an-api-by-using-http-files-in-vscode/

rest-client (vscode) - REST Client allows you to send HTTP request and view the response
in Visual Studio Code directly.
https://marketplace.visualstudio.com/items?itemName=humao.rest-client

postgresql arrays - Работа с массивами в PostgreSQL
https://postgrespro.ru/docs/postgresql/current/arrays