



mtproject-ru

Встреча 2. База данных. Авторизация. DevOps

Дата встречи: 19.02.2026
Формат: офлайн

1. Пользователи
2. Авторизация
3. События
4. 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

```
1 CREATE TABLE IF NOT EXISTS users(
2     id BIGINT GENERATED BY DEFAULT AS IDENTITY (
3         START WITH 100000
4         MINVALUE 100000
5         INCREMENT BY 1
6     ) PRIMARY KEY,
7
8     username VARCHAR(64) UNIQUE NOT NULL,
9     first_name VARCHAR(64) NOT NULL,
10    last_name VARCHAR(64),
11    password VARCHAR(64) NOT NULL
12 );
13
14 CREATE INDEX IF NOT EXISTS users_username_idx ON users(username);
```

```
type UserEntity struct { 3 usages ⚭ lifernod
    Id uint `db:"id"`
    // VARCHAR(64) UNIQUE
    Username string `db:"username"`
    // VARCHAR(64)
    FirstName string `db:"first_name"`
    // VARCHAR(64)
    LastName sql.NullString `db:"last_name"`
    // VARCHAR(64)
    Password string `db:"password"`
}
```

На поле Username создается индекс*

Что там с users(id), много слов?

BIGINT GENERATED BY DEFAULT AS IDENTITY = SERIAL AUTOINCREMENT

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.

База данных. Пользователи

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

Бэкенд. Авторизация

Тут дела обстоят интереснее... 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 refresh умер, то тут 401 (Unauthorized) и пользователю надо будет заново входить в аккаунт. Цикл повторяется (1).

База данных. Авторизация

Ну вот и refresh токен мы будем 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.

База данных. Авторизация

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. Удаление 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

База данных. События

```
type EventEntity struct { 6 usages & lifernod*
    Id          uint      `db:"id"`
    Visibility  Visibility `db:"visibility"`
    // Массив идэй пользователей, которые могут видеть данное событие
    VisibleFor []uint `db:"visible_for"`
    // VARCHAR(128)
    EventTitle string `db:"event_title"`
    // VARCHAR(512)
    EventDescription sql.NullString `db:"event_description"`
    // Если событие начинается в какой то момент и имеет продолжительность
    // (в т.ч. и бесконечную), то указывается дата и время начала события
    EventStart sql.NullTime `db:"event_start"`

    // Если событие ограничено по времени до какой либо даты и какого либо
    // времени, то указывается дата и время окончания события
    //
    // При этом, если начало события указано, то конец события должен быть больше (позже) начала
    EventEnd sql.NullTime `db:"event_end"`

    // Дата и время создания события
    EventCreatedAt time.Time `db:"event_created_at"`
}
```

Репозиторий содержит следующий функционал:

1. Получение события по трем полям:
id, event_created, event_start
2. Обновление события
3. Удаление события
4. Создание события

Домашнее задание для БДшника:

Написать миграцию для создания таблицы.

Самостоятельно выберите формат для EventCreated
(TIMESTAMP или TIMESTAMPZ)

База данных. События

```
type Visibility uint8 5 usages 8 lifernod

const (
    VISIBILITY_PUBLIC = 0 Visibility = iota no usages 8 lifernod
    VISIBLE_PROTECTED = 1 no usages 8 lifernod
    VISIBLE_PRIVATE = 2 no usages 8 lifernod
)

visibility int2 NOT NULL DEFAULT(2),-- VISIBLE_PRIVATE
```

В БД используется int2

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!

База данных. Комнаты

```
type RoomEntity struct { 2 usages & lifernod *
    Id          uint      `db:"id"`
    Visibility  Visibility `db:"visibility"`
    // Массив пользователей, которые могут видеть эту комнату (исключая самих участников)
    VisibleFor []uint `db:"visible_for"`
    // VARCHAR(64)
    RoomTitle string `db:"room_title"`
    // VARCHAR(512)
    RoomDescription sql.NullString `db:"room_description"`
    // Даты, на которых основана команата
    RoomDates []time.Time `db:"room_dates"`
    // Айди пользователя, создавшего комнату
    CreatedBy uint      `db:"created_by"`
    CreatedAt time.Time `db:"created_at"`
}
```

Домашнее задание для БДшника:

Написать миграцию для создания таблицы.

Самостоятельно выберите формат для

RoomCreated (TIMESTAMP или TIMESTAMPZ)

Про RoomDates:

Это массив* дат, которые используются в данной комнате.

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)?

*В postgresql есть массивы, а как с ними работать?

Вот информация на эту тему ([ссылка](#))

База данных. Реляции

```
// UserToEvent Реляция вида many-to-many
Implement interface
type UserToEvent struct { 1 usage & lifernod
    UserId uint `db:"user_id"`
    EventId uint `db:"event_id"`
}
/user_entity.go
```

Дополнения к репозиториям:

Пользователь: привязка события к пользователю, получение всех событий для пользователя

Событие: -

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

```
// RoomToUser реляция вида many-to-many
Implement interface
type RoomToUser struct { 1 usage & lifernod
    RoomId uint `db:"room_id"`
    UserId uint `db:"user_id"`
}
/room_entity.go
```

Реляция располагается рядом с моделью, на название которой она начинается

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

Ход работы

1. БД: пишется модель и интерфейс для репозитория | БЭК: пишутся DTO и мапперы
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 скучно, лучше смотреть страницу расширения в вскоде ([ссылка](#)))

Документирование

POST /api/users/create Создание пользователя

Создание пользователя

Parameters

Name Description

req * required Запрос на создание object Example Value | Model (body)

```
{  
    "password": "string",  
    "username": "string"  
}
```

Parameter content type application/json

Responses

Code Description

200 OK

Example Value | Model

```
{  
    "id": 0,  
    "username": "string"  
}
```

404 Not Found

Example Value | Model

```
"string"
```

Документирование

The screenshot shows the Swagger UI interface for a RESTful API. At the top, there is a navigation bar with the Swagger logo, the URL `/swagger/doc.json`, and a green "Explore" button. Below the navigation bar, there is a green button labeled "OAS 2.0".

The main content area is divided into two sections: "users" and "Models".

users section:

- A green "POST" button followed by the endpoint `/api/users/create` and the description "Создание пользователя".
- A blue "GET" button followed by the endpoint `/api/users/{username}` and the description "Получение пользователя".

Models section:

- A gray box containing the definition for `dto.CreateUserRequest`:

```
dto.CreateUserRequest ▾ {  
    password      ▾ string  
    username      ▾ string  
}
```
- A gray box containing the definition for `dto.UserResponse`:

```
dto.UserResponse ▾ {  
    id            ▾ integer  
    username      ▾ string  
}
```

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

DevOps

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...

*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).

```
1 services:
2   postgres:
3     image: "postgres:18-alpine"
4     container_name: "postgres"
5     profiles:
6       - dev
7       - production
8     environment:
9       POSTGRES_USER: ${POSTGRES_USER}
10      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
11      POSTGRES_DB: ${POSTGRES_DB}
12     ports:
13       - "5432:5432"
14     volumes:
15       - postgres_data:/var/lib/postgres/data
16     networks:
17       - appnet
18
19 pgadmin:
20   image: dpage/pgadmin4:latest
21   container_name: "pgadmin"
22   profiles:
23     - dev
24   environment:
25     PGADMIN_DEFAULT_EMAIL: admin@example.com
26     PGADMIN_DEFAULT_PASSWORD: admin
27   ports:
28     - "8080:80"
29   volumes:
30     - pgadmin_data:/var/lib/pgadmin
31   depends_on:
32     - postgres
33   networks:
34     - appnet
35
36 redis:
37   image: "redis:8-alpine"
38   container_name: "redis"
39   profiles:
40     - dev
41     - production
42   ports:
43     - "6379:6379"
44   volumes:
45     - redis_data:/var/lib/redis
46   networks:
47     - appnet
48
49 redis-insight:
50   image: redis/redisinsight:latest
51   container_name: "redis-insight"
52   profiles:
53     - dev
54   ports:
55     - "5540:5540"
56   volumes:
57     - redis_insight_data:/var/lib/redis_insight
58   depends_on:
59     - redis
60   networks:
61     - appnet
62
63 volumes:
64   pgadmin_data:
65   postgres_data:
66   redis_data:
67   redis_insight_data:
68
69 networks:
70   appnet:
71     driver: bridge
72
73
```

Полезные ссылки

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>