Министерство образования Республики Беларусь

Учреждение образования «ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра «Информационные системы и технологии»

Курсовая работа

по дисциплине «Современные средства разработки серверных приложений»

Разработка REST-сервиса «Гоночное мероприятие»

Выполнил: студент группы 2ИТ-12

Бондарев Е. А.

Проверил: Заведующий кафедрой ИСиТ

Казаков В. Е.

Витебск, 2024

СОДЕРЖАНИЕ

Введение…………………………………………………………………………....…2

1. Разработка плана для решения поставленной задачи……………………...……3

2. Сервис и его структура…….………………………………………………............4

3. Тестирование…………………………………………………………………...…23

3.1 Авторизация и регистрация пользователей…………………………..…23

3.2 Тестирование в роли «USER»……………………………………….…...24

3.3 Тестирование в роли «ADMIN»………………………………………....26

Заключение………………………………………………………………........27

**Введение**

В современном информационном обществе важным направлением развития является создание эффективных и масштабируемых веб-приложений, способных обеспечить удобный доступ к данным и функциональности через сеть Интернет. Одним из ключевых компонентов в разработке таких приложений является REST-архитектура, которая представляет собой стандарт для построения веб-сервисов, обеспечивающих простоту, надежность и гибкость взаимодействия между клиентами и серверами.

Целью данной курсовой работы является исследование, проектирование и разработка REST-сервера, который является основой для построения современных веб-приложений. В рамках данного исследования будет рассмотрено не только теоретическое основание REST-архитектуры, но и практические аспекты реализации RESTful API. Анализ основных принципов работы REST-сервисов, выбор инструментов и технологий для создания RESTful сервера, а также процесс проектирования и разработки RESTful API будут освещены в данной работе.

Изучение и практическое применение REST-сервера позволит не только углубить знания в области веб-разработки, но и приобрести практические навыки в создании гибких и масштабируемых веб-сервисов. Результаты данного исследования будут способствовать лучшему пониманию принципов работы REST-сервисов и их применению в современной веб-разработке, а также предоставят практические рекомендации по созданию высокопроизводительных и надежных RESTful API.

**1. Разработка плана для решения поставленной задачи**

Тщательное планирование служит основой для создания качественного программного продукта. REST-сервис “гоночное мероприятие” должен выступать в роли всесторонней платформы для координации и управления участниками, спонсорами и зрителями гоночных соревнований. В рамках этого сервиса предусмотрена функциональность для регистрации и авторизации пользователей, а также система прав доступа к информации о гонках, гонщиках, спонсорах и зрителях.

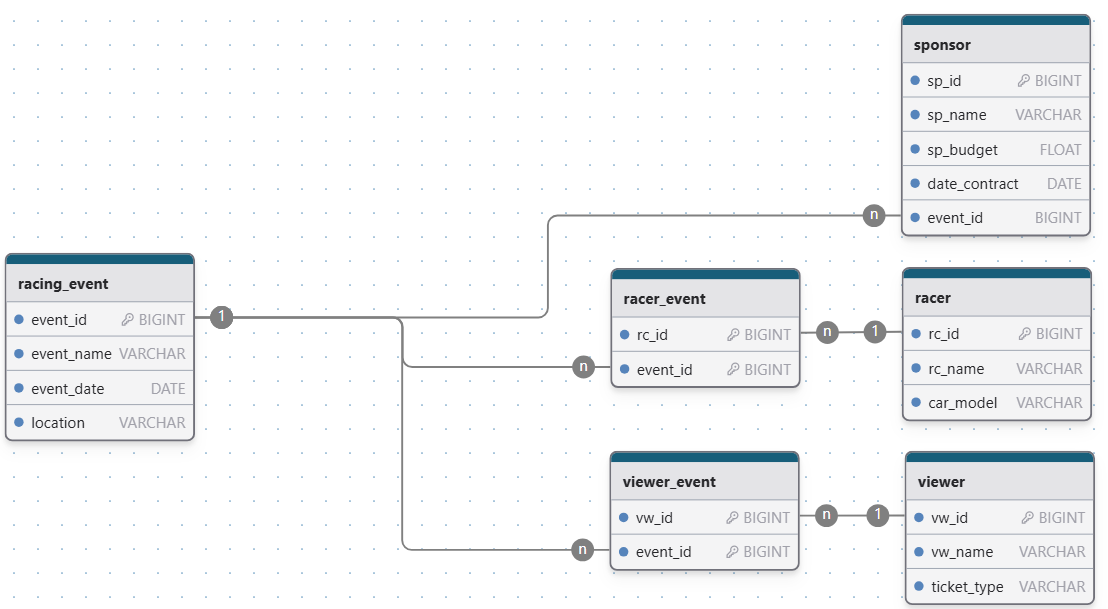
Для начала нужно построить взаимосвязь между сущностями в сервисе. Для этого я собрал схему базы данных моего проекта:  


Рис. 2 – схема базы данных

Более подробное описание функционала сервиса:

1. **Методы работы с участниками:**

* Создание гонщика – (/racer/create) – POST
* Получение всех гонщиков – (/racer) – GET
* Получение гонщиков по id мероприятия – (/racer/event/{id}) – GET
* Получение гонщика по его id – (/racer/{id}) – GET
* Получение гонщика по его имени – (/racer/name/{name}) – GET
* Получение гонщиков по модели автомобиля – (/racer/carModel/{carModel}) – GET
* Редактирование гонщика – (/racer/edit) – PUT
* Удаление гонщика по его id – (/racer/{id}) – DELETE

1. **Методы работы со спонсорами:**

* Создание спонсора – (/sponsor/create) – CREATE
* Получение всех спонсоров – (/sponsor) – GET
* Получение спонсора по id мероприятия – (/sponsor/event/{id}) – GET
* Получение спонсора по его id – (/sponsor/{id}) – GET
* Получение спонсора по его имени – (/sponsor/name/{name}) – GET
* Редактирование спонсора – (/sponsor/edit) – PUT
* Удаление спонсора – (/sponsor/{id}) – DELETE

1. **Методы работы со зрителями:**

* Создание зрителя – (/viewer/create) – POST
* Получение всех зрителей – (/viewer) – GET
* Получение зрителей по id мероприятия – (/viewer/event/{id}) – GET
* Получение зрителя по его id – (/viewer/{id}) – GET
* Получение зрителя по его имени – (/viewer/name/{name}) – GET
* Получение зрителей по типу билета(Normal/VIP) – (/viewer) – GET
* Редактирование зрителя – (/viewer/edit) – PUT
* Удаление зрителя – (/viewer/{id}) – DELETE

1. **Методы работы с мероприятиями:**

* Создание мероприятия – (/racingEvent/create) – POST
* Получение всех мероприятий – (/racingEvent) – GET
* Получение мероприятия по названию – (/racingEvent/name/{name}) – GET
* Получение мероприятия по его id – (/racingEvent/{id}) – GET
* Получение зрителей по типу билета – (/racingEvent/{id}/viewersByTicket/{ticketType}) – GET
* Получение суммы бюджетов спонсоров – (/racingEvent/{id}/sponsor-budgets-sum) – GET
* Получение спонсоров, которые заключили контракт в течение определенного количества месяцев до начала мероприятия – (/racingEvent/{id}/sponsorContract/{month}) – GET
* Удаление мероприятия - (/racingEvent/{id}) – DELETE

**2. Сервис и его структура**

Для работы с данными была выбрана MySQL Workbench. MySQL Workbench — это официальная интегрированная среда разработки (IDE), созданная специально для MySQL корпорацией Oracle.

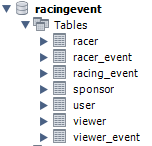


Рис. 2 – таблицы базы данных

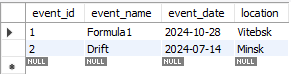


Рис. 3 – таблица racing\_event

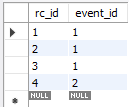
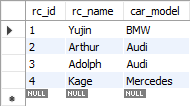
****

Рис. 4 – таблица racer и связывающая таблица racer\_event для реализации связи Many To Many

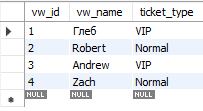
****

Рис. 5 – таблица viewer и связывающая таблица viewer\_event для реализации связи Many To Many

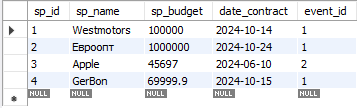
****

Рис. 6 – таблица sponsor

**racing\_event** – таблица, в которой хранятся участники, зрители и спонсоры. В таблице имеются следующие поля:

* event\_id – id мероприятия
* event\_name – описание товара
* event\_date – название товара
* location – количество товара

**racer** – таблица, в которой хранятся все гонщики. В таблице имеются следующие поля:

* rc\_id – id гонщика
* rc\_name – имя гонщика
* carModel – модель автомобиля

**racer\_event** – таблица, которая связывает гонщика и мероприятие, так как гонщик может участвовать в нескольких мероприятиях, и мероприятие может иметь несколько гонщиков. В таблице имеются следующие поля:

* rc\_id – id гонщика
* event\_id – id мероприятия

**viewer** – таблица, в которой хранятся все зрители. В таблице имеются следующие поля:

* vw\_id – id зрителя
* vw\_name – имя зрителя
* ticketType – тип билета

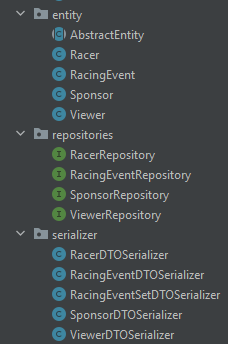
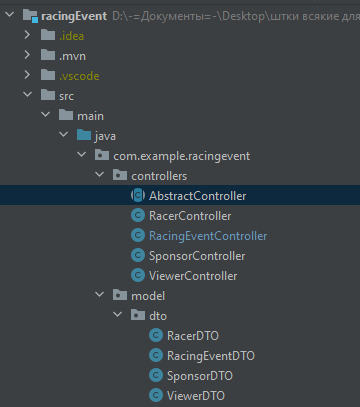
**viewer\_event** – таблица, которая связывает зрителя и мероприятие, так как зритель может купить билеты на несколько мероприятий, и мероприятие может иметь несколько зрителей. В таблице имеются следующие поля:

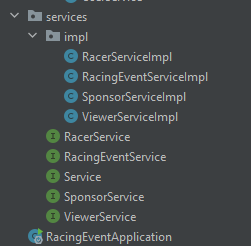
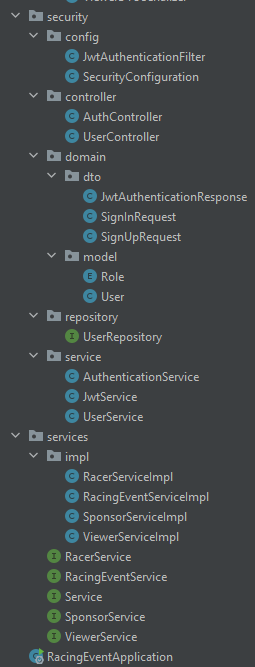
* vw\_id – id зрителя
* event\_id – id мероприятия

**sponsor** – таблица, в которой хранятся все спонсоры. В таблице имеются следующие поля:

* sp\_id – id спонсора
* sp\_name – имя спонсора
* sp\_budget– бюджет спонсора
* date\_contract – дата заключения контракта
* event\_id – id мероприятия

Архитектура проекта выглядит следующим образом:





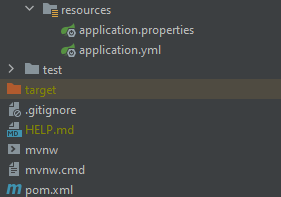


Рис. 7 – архитектура проекта

Пакет **com.example.racingevent.controllers**

1. Абстрактный класс-предок **AbstractController**

Абстрактный контроллер, который использует обобщенный тип T, расширяющий AbstractEntity. Содержит основные методы GET (список сущностей, по id и по имени) и DELETE. Вот краткое описание каждого метода:

* init: Этот метод инициализирует заголовки HTTP, которые будут использоваться в ответах контроллера.
* get: Этот метод возвращает список всех сущностей. Если список пуст, возвращается статус HttpStatus.NOT\_FOUND.
* getById: Этот метод возвращает сущность по ее идентификатору. Если сущность не найдена, возвращается статус HttpStatus.NOT\_FOUND.
* getByName: Этот метод возвращает список сущностей с определенным именем. Если список пуст, возвращается статус HttpStatus.NOT\_FOUND.
* delete: Этот метод удаляет сущность по ее идентификатору. Возвращается статус HttpStatus.OK

1. Класс **RacerController**

Этот контроллер отвечает за операции, проводимые над сущностью Racer, а также расширяет абстрактный контроллер.

* getRacersByEvent: Этот метод возвращает список гонщиков, участвующих в определенном гоночном событии. Если список пуст, возвращается статус HttpStatus.NOT\_FOUND.
* getRacersByCarModel: Этот метод возвращает список гонщиков, у которых определенная модель автомобиля. Если список пуст, возвращается статус HttpStatus.NOT\_FOUND.
* createRacer: Этот метод создает нового гонщика. Если при сохранении возникает исключение, возвращается сообщение об ошибке и статус HttpStatus.NOT\_FOUND.
* updateRacer: Этот метод обновляет существующего гонщика. Если при обновлении возникает исключение, возвращается сообщение об ошибке и статус HttpStatus.NOT\_FOUND.

1. Класс **RacigEventController**

Этот контроллер отвечает за операции, проводимые над сущностью RacingEvent, а также расширяет абстрактный класс AbstractEntity.

* createEvent: Этот метод создает новое гоночное событие. Если при сохранении возникает исключение, возвращается сообщение об ошибке и статус HttpStatus.NOT\_FOUND.
* getEventViewersByTicketType: Этот метод возвращает список зрителей определенного гоночного события с определенным типом билета.
* getSponsorBudgetsSum: Этот метод возвращает сумму бюджетов всех спонсоров определенного гоночного события.
* getSponsorsContractedBeforeEvent: Этот метод возвращает список спонсоров, которые заключили контракт до определенного гоночного события.

1. Класс **SponsorController**

Этот контроллер отвечает за операции, проводимые над сущностью Sponsor, а также расширяет абстрактный контроллер.

* getSponsorsByEvent: Этот метод возвращает список спонсоров, связанных с определенным гоночным событием. Если список пуст, возвращается статус HttpStatus.NOT\_FOUND.
* createSponsor: Этот метод создает нового спонсора. Если при сохранении возникает исключение, возвращается сообщение об ошибке и статус HttpStatus.NOT\_FOUND.
* updateSponsor: Этот метод обновляет существующего спонсора. Если при обновлении возникает исключение, возвращается сообщение об ошибке и статус HttpStatus.NOT\_FOUND.

1. Класс **ViewerController**

Этот контроллер отвечает за операции, проводимые над сущностью Viewer, а также расширяет абстрактный контроллер.

* getViewersByEvent: Этот метод возвращает список зрителей, связанных с определенным гоночным событием. Если список пуст, возвращается статус HttpStatus.NOT\_FOUND.
* getViewersByTicketType: Этот метод возвращает список зрителей с определенным типом билета. Если список пуст, возвращается статус HttpStatus.NOT\_FOUND.
* createViewer: Этот метод создает нового зрителя. Если при сохранении возникает исключение, возвращается сообщение об ошибке и статус HttpStatus.NOT\_FOUND.
* updateViewer: Этот метод обновляет существующего зрителя. Если при обновлении возникает исключение, возвращается сообщение об ошибке и статус HttpStatus.NOT\_FOUND.

Пакет **com.example.racingevent.model**

**.dto**

**RacerDTO**, **RacingEventDTO**, **SponsorDTO**, **ViewerDTO**: Это классы DTO для соответствующих сущностей. Они содержат поля, которые соответствуют полям сущностей и используются для передачи данных между подсистемами проекта. Я решил их создать для предотвращения цикличного вывода информации в JSCON (Например, мне нужно будет вывести ивенты, которые посетит один зритель, и мне выдадут информацию об ивентах, а также зрителях, спонсорах и гонщиках этих ивентов)

**.entity**

1. Абстрактный класс-предок **AbstractEntity**

Этот класс является базовым классом для сущностей и обеспечивает общую функциональность, такую как идентификация и сравнение объектов.

* Аннотация @MappedSuperclass: Эта аннотация JPA указывает, что класс является базовым классом и его поля будут включены в классы-наследники, которые являются сущностями.
* Поле класса: В классе есть одно поле id, которое представляет собой идентификатор сущности. Это поле помечено аннотациями @Id и @GeneratedValue, которые указывают, что это поле является первичным ключом и его значение генерируется автоматически.
* Методы hashCode() и equals(Object obj): Эти методы переопределены для обеспечения правильного сравнения экземпляров AbstractEntity. Метод hashCode() возвращает хеш-код, основанный на id, а метод equals(Object obj) сравнивает два объекта на равенство, основываясь на их id.

1. Класс **Racer**
   * + Аннотация @Entity: Эта аннотация указывает, что этот класс является сущностью JPA.
     + Аннотация @Table(name = "racer"): Эта аннотация указывает, что экземпляры этого класса будут сохраняться в таблице с именем “racer”.
     + Аннотация @AttributeOverride(name = "id", column = @Column(name = "rc\_id")): Эта аннотация указывает, что поле id будет сохраняться в столбце с именем “rc\_id”.
     + Наследование от AbstractEntity: Класс наследуется от AbstractEntity, что означает, что он наследует все поля и методы этого класса, включая поле id.
     + name, carModel: Это поля, представляющие имя гонщика и модель его автомобиля.
     + rcEvents: Это поле представляет гоночные события, в которых участвует гонщик. Оно аннотировано как @ManyToMany, что указывает на отношение “многие ко многим” между гонщиками и гоночными событиями.
2. Класс **RacingEvent**
   * + @Entity: Эта аннотация указывает, что класс является сущностью JPA (Java Persistence API), которая может быть отображена на таблицу в базе данных.
     + @Table(name = “racing\_event”): Эта аннотация указывает, что экземпляры этого класса будут сохраняться в таблице с именем “racing\_event”.
     + @AttributeOverride(name = “id”, column = @Column(name = “event\_id”)): Эта аннотация указывает, что поле id, унаследованное от AbstractEntity, будет сохраняться в столбце с именем “event\_id”.
     + Наследование от AbstractEntity;
     + eventName, date, location: Это поля, представляющие имя события, дату и место проведения. Они аннотированы как @Column, что указывает, что они будут сохраняться в столбцах с соответствующими именами.
     + racers, viewers, sponsors: Это поля, представляющие гонщиков, зрителей и спонсоров, связанных с гоночным событием. Они аннотированы как @ManyToMany и @OneToMany, что указывает на отношения “многие ко многим” и “один ко многим” между гоночным событием и гонщиками, зрителями и спонсорами соответственно.
3. Класс **Sponsor**

* @Entity: Эта аннотация указывает, что класс является сущностью JPA.
* @Table(name = "sponsor"): Эта аннотация указывает, что экземпляры этого класса будут сохраняться в таблице с именем “sponsor”.
* @AttributeOverride(name = "id", column = @Column(name = "sp\_id")): Эта аннотация указывает, что поле id, унаследованное от AbstractEntity, будет сохраняться в столбце с именем “sp\_id”.
* Наследование от AbstractEntity;
* spName, budget, dateOfContract: Это поля, представляющие имя спонсора, его бюджет и дату заключения контракта.
* spEvents: Это поле представляет гоночное событие, которое спонсирует спонсор. Оно аннотировано как @ManyToOne, что указывает на отношение “многие к одному” между спонсорами и гоночными событиями.

1. Класс **Viewer**

* @Entity: Эта аннотация указывает, что класс является сущностью JPA.
* @Table(name = "viewer"): Эта аннотация указывает, что экземпляры этого класса будут сохраняться в таблице с именем “viewer”.
* @AttributeOverride(name = "id", column = @Column(name = "vw\_id")): Эта аннотация указывает, что поле id, унаследованное от AbstractEntity, будет сохраняться в столбце с именем “vw\_id”.
* Наследование от AbstractEntity;
* vwName, ticketType: Это поля, представляющие имя зрителя и тип его билета.
* vwEvents: Это поле представляет гоночные события, которые посещает зритель. Оно аннотировано как @ManyToMany, что указывает на отношение “многие ко многим” между зрителями и гоночными событиями.

**.repositories**

1. Интерфейс **RacerRepository**:

* Этот интерфейс предоставляет методы для работы с гонщиками в базе данных. Это включает в себя поиск гонщиков по имени, модели автомобиля и гоночному событию.
* @Repository: Эта аннотация Spring указывает, что класс является репозиторием, т.е. он будет использоваться для операций с базой данных.
* Расширение JpaRepository: Этот интерфейс Spring Data JPA предоставляет набор стандартных методов JPA для работы с базой данных. RacerRepository расширяет JpaRepository для сущности Racer с идентификатором типа Long.

1. Интерфейс **RacingEventRepository**:

* Этот интерфейс предоставляет методы для работы с гоночными событиями в базе данных. Это включает в себя поиск гоночных событий по имени.
* @Repository: Эта аннотация Spring указывает, что класс является репозиторием, т.е. он будет использоваться для операций с базой данных.
* Расширение JpaRepository: Этот интерфейс Spring Data JPA предоставляет набор стандартных методов JPA для работы с базой данных. RacingEventRepository расширяет JpaRepository для сущности RacingEvent с идентификатором типа Long.

1. Интерфейс **RacerRepository**:

* Этот интерфейс предоставляет методы для работы со спонсорами в базе данных. Это включает в себя поиск спонсоров по имени и гоночному событию.
* @Repository: Эта аннотация Spring указывает, что класс является репозиторием, т.е. он будет использоваться для операций с базой данных.
* Расширение JpaRepository: Этот интерфейс Spring Data JPA предоставляет набор стандартных методов JPA для работы с базой данных. SponsorRepository расширяет JpaRepository для сущности Sponsor с идентификатором типа Long.

1. Интерфейс **RacerRepository**:

* Этот интерфейс предоставляет методы для работы с зрителями в базе данных. Это включает в себя поиск зрителей по имени, типу билета и гоночному событию.
* @Repository: Эта аннотация Spring указывает, что класс является репозиторием, т.е. он будет использоваться для операций с базой данных.
* Расширение JpaRepository: Этот интерфейс Spring Data JPA предоставляет набор стандартных методов JPA для работы с базой данных. ViewerRepository расширяет JpaRepository для сущности Viewer с идентификатором типа Long.

**.serializer**

1. Класс **RacerDTOSerializer**:

Этот класс является сериализатором для сущности Racer. Он наследует StdSerializer<Set<Racer>> и переопределяет метод serialize, который преобразует набор сущностей Racer в набор DTO и затем сериализует их в формат JSON.

1. Класс **RacingEventDTOSerializer**:

Этот класс является сериализатором для сущности RacingEvent. Он наследует StdSerializer<RacingEvent> и переопределяет метод serialize, который преобразует сущность RacingEvent в DTO и затем сериализует его в формат JSON.

1. Класс **RacingEventSetDTOSerializer**:

Этот класс является сериализатором для набора сущностей RacingEvent. Он наследует StdSerializer<Set<RacingEvent>> и переопределяет метод serialize, который преобразует набор сущностей RacingEvent в набор DTO и затем сериализует их в формат JSON.

1. Класс **SponsorDTOSerializer**:

Этот класс является сериализатором для сущности Sponsor. Он наследует StdSerializer<Set<Sponsor>> и переопределяет метод serialize, который преобразует набор сущностей Sponsor в набор DTO и затем сериализует их в формат JSON.

1. Класс **ViewerDTOSerializer**:

Этот класс является сериализатором для сущности Viewer. Он наследует StdSerializer<Set<Viewer>> и переопределяет метод serialize, который преобразует набор сущностей Viewer в набор DTO и затем сериализует их в формат JSON.

Пакет **com.example.racingevent.services**

1. Абстрактный интерфейс **Service**:

* Этот интерфейс предоставляет базовые операции CRUD для работы с сущностями. Это включает в себя чтение сущности по идентификатору, чтение всех сущностей, сохранение сущности, удаление сущности по идентификатору и редактирование сущности.
* Параметризованный интерфейс Service является параметризованным интерфейсом, где T является подклассом AbstractEntity. Это означает, что он может работать с любым типом объекта, который является подклассом AbstractEntity

1. Интерфейс **RacerService**:
   * + Этот интерфейс предоставляет методы для работы с гонщиками, включая чтение гонщиков по идентификатору события, имени и модели автомобиля, а также все базовые операции CRUD, унаследованные от Service.
     + Расширение Service: RacerService расширяет интерфейс Service с параметром Racer. Это означает, что он наследует все методы Service и может работать с объектами типа Racer.
2. Интерфейс **RacingEventService**:

* Этот интерфейс предоставляет методы для работы с гоночными событиями, включая чтение событий по имени, получение зрителей события по типу билета, получение суммы бюджетов спонсоров и получение спонсоров, заключивших контракт до события, а также все базовые операции CRUD, унаследованные от Service.
* Расширение Service: RacingEventService расширяет интерфейс Service с параметром RacingEvent. Это означает, что он наследует все методы Service и может работать с объектами типа RacingEvent.

1. Интерфейс **SponsorService**:

* Этот интерфейс предоставляет методы для работы со спонсорами, включая чтение спонсоров по идентификатору события и имени, а также все базовые операции CRUD, унаследованные от Service.
* Расширение Service: SponsorService расширяет интерфейс Service с параметром Sponsor. Это означает, что он наследует все методы Service и может работать с объектами типа Sponsor.

1. Интерфейс **ViewerService**:

* Этот интерфейс предоставляет методы для работы с зрителями, включая чтение зрителей по идентификатору события, имени и типу билета, а также все базовые операции CRUD, унаследованные от Service.
* Расширение Service: ViewerService расширяет интерфейс Service с параметром Viewer. Это означает, что он наследует все методы Service и может работать с объектами типа Viewer.

**.impl**

1. Класс RacerServiceImpl:

* Методы: Этот класс предоставляет методы для основных CRUD-операций (создание, чтение, обновление, удаление), а также чтение гонщиков по идентификатору события, имени и модели автомобиля.
* Аннотация @Service: Этот класс помечен аннотацией @Service, что означает, что Spring будет управлять его жизненным циклом как сервисный компонент.
* Реализация интерфейса RacerService: Этот класс реализует интерфейс RacerService, что означает, что он обязан реализовать все методы, определенные в этом интерфейсе. Это включает в себя методы для чтения гонщиков по идентификатору события, имени и модели автомобиля, а также основные CRUD-операции.

1. Класс **RacingServiceImpl**:

* Методы: Этот класс предоставляет методы для основных CRUD-операций (создание, чтение, обновление, удаление), а также чтение гоночных событий по имени, получение суммы бюджетов спонсоров, получение спонсоров, заключивших контракт до события, и получение зрителей события по типу билета.
* Аннотация @Service: Этот класс помечен аннотацией @Service, что означает, что Spring будет управлять его жизненным циклом как сервисный компонент.
* Реализация интерфейса RacingEventService: Этот класс реализует интерфейс RacingEventService, что означает, что он обязан реализовать все методы, определенные в этом интерфейсе. Это включает в себя методы для чтения гоночных событий по имени, получения суммы бюджетов спонсоров, получения спонсоров, заключивших контракт до события, и получения зрителей события по типу билета, а также основные CRUD-операции.

1. Класс **SponsorServiceImpl**:

* Методы: Этот класс предоставляет методы для основных CRUD-операций (создание, чтение, обновление, удаление), а также чтение спонсоров по идентификатору события и имени.
* Аннотация @Service: Этот класс помечен аннотацией @Service, что означает, что Spring будет управлять его жизненным циклом как сервисный компонент.
* Реализация интерфейса SponsorService: Этот класс реализует интерфейс SponsorService, что означает, что он обязан реализовать все методы, определенные в этом интерфейсе. Это включает в себя методы для чтения спонсоров по идентификатору события и имени, а также основные CRUD-операции.

1. Класс **ViewerServiceImpl**:

* Методы: Этот класс предоставляет методы для основных CRUD-операций (создание, чтение, обновление, удаление), а также чтение зрителей по идентификатору события, имени и типу билета.
* Аннотация @Service: Этот класс помечен аннотацией @Service, что означает, что Spring будет управлять его жизненным циклом как сервисный компонент.
* Реализация интерфейса ViewerService: Этот класс реализует интерфейс ViewerService, что означает, что он обязан реализовать все методы, определенные в этом интерфейсе. Это включает в себя методы для чтения зрителей по идентификатору события, имени и типу билета, а также основные CRUD-операции.

Пакет **com.example.racingevent.security**

**.config**

1. Класс **JwtAuthenticationFilter**:

* Аннотация @Component: Этот класс помечен аннотацией @Component, что означает, что Spring будет управлять его жизненным циклом как компонентом.
* Аннотация @RequiredArgsConstructor: Эта аннотация генерирует конструктор для всех final полей, которые были инициализированы при объявлении, и для всех полей с аннотацией @NonNull. В данном случае, это jwtService и userService.
* Расширение OncePerRequestFilter: Этот класс расширяет OncePerRequestFilter, что означает, что он будет применяться один раз за каждый запрос HTTP.
* Методы класса JwtAuthenticationFilter:

doFilterInternal (HttpServletRequest request, HttpServletResponse response, FilterChain filterChain): Этот метод выполняет фильтрацию запросов. Он извлекает токен JWT из заголовка запроса, проверяет его валидность и, если токен валиден, аутентифицирует пользователя.

1. Класс **SecurityConfiguration**:

* Этот класс является конфигурационным и управляет настройками безопасности в вашем приложении.
* Он использует JwtAuthenticationFilter и UserService, которые внедряются через конструктор.
* Метод securityFilterChain(HttpSecurity http) настраивает цепочку фильтров безопасности, включая отключение CSRF, разрешение CORS, настройку доступа к конечным точкам и добавление фильтра аутентификации JWT.
* Метод passwordEncoder() возвращает кодировщик паролей, который использует BCrypt.
* Метод authenticationProvider() возвращает провайдер аутентификации, который использует UserService для загрузки пользовательских данных и кодировщик паролей для сравнения паролей.
* Метод authenticationManager(AuthenticationConfiguration config) возвращает менеджер аутентификации, который используется для обработки аутентификации.

**.controller**

1. Класс **AuthController**:

* Аннотация @RestController: Эта аннотация указывает, что данный класс является контроллером в модели MVC и его результаты напрямую записываются в тело ответа, а не возвращают имя представления.
* Аннотация @RequestMapping("/auth"): Эта аннотация указывает, что все методы в этом контроллере будут обрабатывать запросы, начинающиеся с /auth.
* Аннотация @RequiredArgsConstructor: Эта аннотация Lombok генерирует конструктор для всех final полей, которые были инициализированы при объявлении, и для всех полей с аннотацией Аннотация @NonNull. В данном случае, это authenticationService.
* Поле authenticationService: Это поле представляет сервис для аутентификации пользователей. Он внедряется через конструктор.
* Метод signUp(@RequestBody SignUpRequest request): Этот метод обрабатывает POST-запросы по пути /auth/sign-up. Он принимает объект SignUpRequest в теле запроса и возвращает ответ JwtAuthenticationResponse, который генерируется сервисом аутентификации.
* Метод signIn(@RequestBody SignInRequest request): Этот метод обрабатывает POST-запросы по пути /auth/sign-in. Он принимает объект SignInRequest в теле запроса и возвращает ответ JwtAuthenticationResponse, который генерируется сервисом аутентификации.

1. Класс **UserController**:

* Аннотация @RestController: Эта аннотация указывает, что данный класс является контроллером в модели MVC и его результаты напрямую записываются в тело ответа, а не возвращают имя представления.
* Аннотация @RequestMapping("/user"): Эта аннотация указывает, что все методы в этом контроллере будут обрабатывать запросы, начинающиеся с /user.
* Полe userService: Это поле представляет сервис для работы с пользователями. Он внедряется через конструктор.
* Полe headers: Это поле представляет заголовки HTTP, которые будут использоваться при создании ответов.
* Метод init(): Этот метод инициализирует заголовки HTTP, устанавливая тип контента как APPLICATION\_JSON.
* Метод get(): Этот метод обрабатывает GET-запросы по пути /user. Он возвращает список всех пользователей или статус NOT\_FOUND, если список пользователей пуст.
* Метод getAdmin(): Этот метод обрабатывает PUT-запросы по пути /user/get-admin. Он вызывает метод getAdmin() из userService.
* Метод getService(): Этот метод возвращает userService.

**.domain**

**.dto**

* Класс **JwtAuthenticationResponse**: Этот класс представляет собой ответ на аутентификацию JWT. Он содержит одно поле token, которое представляет собой JWT токен. Аннотации @Data, @Builder, @NoArgsConstructor и @AllArgsConstructor генерируют геттеры, сеттеры, конструктор без параметров и конструктор со всеми параметрами соответственно.
* Класс **SignInRequest**: Этот класс представляет собой запрос на вход в систему. Он содержит два поля: username и password, которые представляют собой имя пользователя и пароль соответственно. Аннотация @Data генерирует геттеры и сеттеры для этих полей.
* Класс **SignUpRequest**: Этот класс представляет собой запрос на регистрацию. Он также содержит два поля: username и password, которые представляют собой имя пользователя и пароль соответственно. Аннотации @Data, @NoArgsConstructor и @AllArgsConstructor генерируют геттеры, сеттеры, конструктор без параметров и конструктор со всеми параметрами соответственно.

**.model**

1. Перечисление **User**:

* Это перечисление, которое определяет две роли: ROLE\_USER и ROLE\_ADMIN. Эти роли могут быть использованы для определения уровня доступа пользователя в приложении.

1. Класс **User**:

Этот класс представляет собой сущность, которая описывает пользователя. Он содержит информацию о пользователе, такую как его имя, пароль и роль. Класс используется для аутентификации пользователя в Spring Security.

* Аннотация @Entity: Эта аннотация указывает, что данный класс является сущностью в контексте JPA.
* Аннотация @Table: Эта аннотация JPA указывает, что объекты этого класса будут сохраняться в таблице “user”.
* Реализация интерфейса UserDetails: Этот интерфейс предоставляет основную информацию о пользователе для Spring Security.
* Методы getAuthorities(), isAccountNonExpired(), isAccountNonLocked(), isCredentialsNonExpired() и isEnabled(): Эти методы переопределены из интерфейса UserDetails и возвращают информацию о пользователе, необходимую для Spring Security.

**.repository**

Интерфейс **UserRepository**:

* Этот интерфейс предоставляет методы для работы с пользователями в базе данных. Это включает в себя поиск пользователей по имени пользователя и проверку существования пользователя с данным именем пользователя.
* Аннотация @Repository: Эта аннотация указывает, что данный интерфейс является репозиторием, то есть механизмом для инкапсуляции хранения, извлечения и поиска объектов.
* Расширение JpaRepository: Этот интерфейс расширяет JpaRepository для сущности User с идентификатором типа Long. Это означает, что он наследует набор базовых методов JPA для выполнения операций CRUD (создание, чтение, обновление, удаление) над сущностями User.

**.service**

1. Класс **AuthenticationService**:

Этот класс является сервисом, который обрабатывает аутентификацию и регистрацию пользователей.

* Аннотация @Service: Эта аннотация указывает, что данный класс является сервисом в контексте Spring.
* Аннотация @RequiredArgsConstructor: Эта аннотация Lombok генерирует конструктор для всех final полей, которые были инициализированы при объявлении, и для всех полей с аннотацией @NonNull.
* Поле userService: Это поле представляет сервис для работы с пользователями. Он внедряется через конструктор.
* Поле jwtService: Это поле представляет сервис для работы с JWT. Он внедряется через конструктор.
* Поле passwordEncoder: Это поле представляет кодировщик паролей. Он внедряется через конструктор.
* Поле authenticationManager: Это поле представляет менеджер аутентификации. Он внедряется через конструктор.
* Метод signUp(SignUpRequest request): Этот метод обрабатывает регистрацию пользователей. Он принимает объект SignUpRequest, создает нового пользователя, сохраняет его в базе данных и возвращает JWT токен.
* Метод signIn(SignInRequest request): Этот метод обрабатывает аутентификацию пользователей. Он принимает объект SignInRequest, аутентифицирует пользователя и возвращает JWT токен.

1. Класс **JwtService**:

Этот класс является сервисом, который обрабатывает операции с JWT токенами, такие как их генерация, валидация и извлечение данных.

* Аннотация @Service: Эта аннотация указывает, что данный класс является сервисом в контексте Spring.
* Поле jwtSigningKey: Это поле представляет ключ для подписи JWT токенов.
* Метод extractUserName(String token): Этот метод извлекает имя пользователя из JWT токена.
* Метод generateToken(UserDetails userDetails): Этот метод генерирует JWT токен для указанных пользовательских данных.
* Метод isTokenValid(String token, UserDetails userDetails): Этот метод проверяет, является ли JWT токен валидным для указанных пользовательских данных.
* Метод extractClaim(String token, Function<Claims, T> claimsResolvers): Этот метод извлекает данные из JWT токена.
* generateToken(Map<String, Object> extraClaims, UserDetails userDetails): Этот метод генерирует JWT токен с дополнительными данными для указанных пользовательских данных.
* Метод isTokenExpired(String token): Этот метод проверяет, просрочен ли JWT токен.
* Метод extractExpiration(String token): Этот метод извлекает дату истечения JWT токена.
* Метод extractAllClaims(String token): Этот метод извлекает все данные из JWT токена.
* Метод getSigningKey(): Этот метод возвращает ключ для подписи JWT токенов.

1. Класс **UserService**:

Этот класс является сервисом, который обрабатывает операции с пользователями, такие как их создание, получение, сохранение и обновление

* Аннотация @Service: Эта аннотация указывает, что данный класс является сервисом в контексте Spring.
* Полe repository: Это поле представляет репозиторий для работы с пользователями. Оно внедряется через конструктор.
* Метод read(): Этот метод возвращает список всех пользователей.
* Метод save(User user): Этот метод сохраняет пользователя в базе данных.
* Метод create(User user): Этот метод создает нового пользователя, если пользователя с таким именем еще не существует.
* Метод getByUsername(String username): Этот метод возвращает пользователя по имени пользователя.
* Метод userDetailsService(): Этот метод возвращает службу деталей пользователя.
* Метод getCurrentUser(): Этот метод возвращает текущего пользователя, используя контекст Spring Security.
* Метод getAdmin(): Этот метод обновляет роль текущего пользователя на ROLE\_ADMIN.

**3. Тестирование**

* 1. **Регистрация и авторизация пользователей**

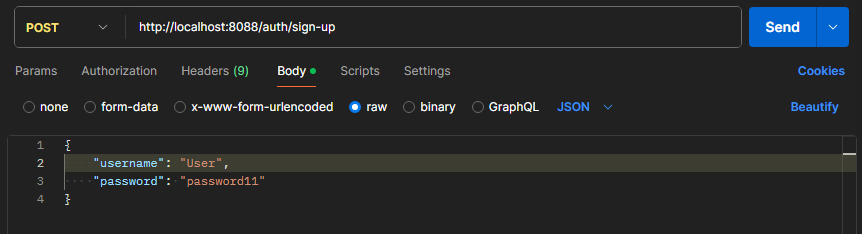
Для начала нужно зарегистрировать пользователя (т.е. создать). Для этого без авторизации отправляем POST запрос:  


Рис. 8 – Запрос для регистрации

При успешной регистрации мы получим уникальный токен:

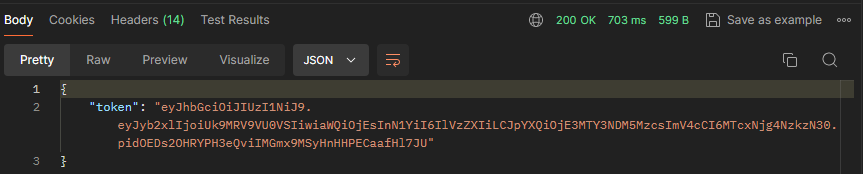


Рис. 9 – Ответ на регистрацию в виде уникального токена

А теперь попробуем авторизоваться

Если хотя бы один элемент (username или password) не соответствует существующему пользователю, то система в ответ вернёт HTTP-статус 403

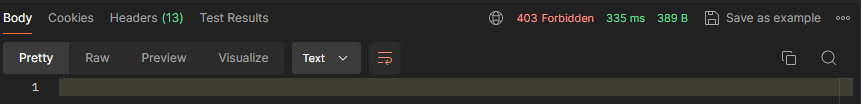


Рис. 10 – Ответ на авторизацию с неверными данными

А при успешной авторизации в ответ вернёт уникальный токет с HTTP-статусом 200:

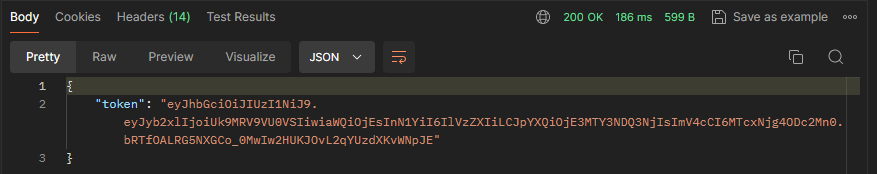


Рис. 11 – Ответ на авторизацию с верными данными

* 1. **Тестирование в роли «USER»**

Для получения доступа к запросам, требующим роль USER, необходимо авторизовать его благодаря полученному при регистрации/авторизации токену:

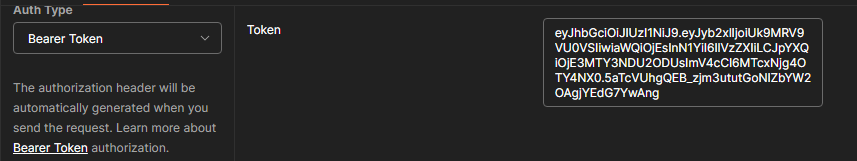


Рис. 12 – Установка токена

Теперь у нас есть доступ к запросам для данной роли. В качестве примера приведу выполнение GET запросов:

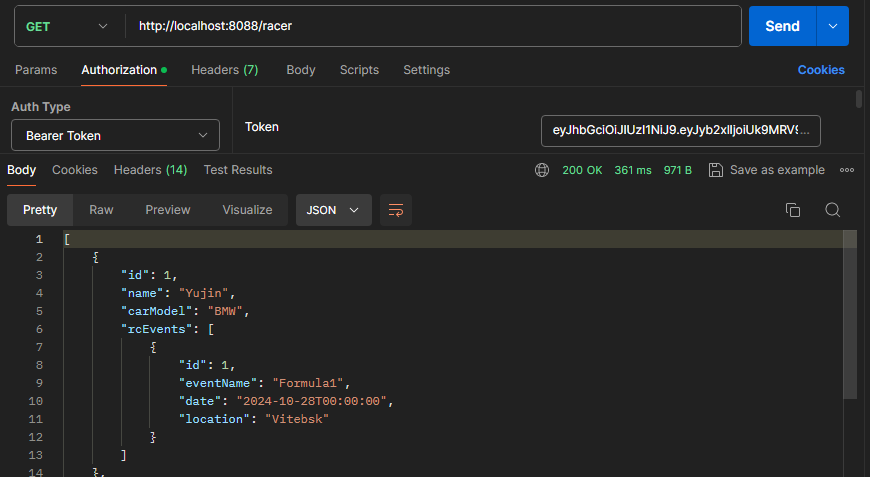


Рис. 13 – Получение всех гонщиков

**Тестирование эндпоинтов:  
1. Получение зрителей с определённым типом билета определённого ивента:**

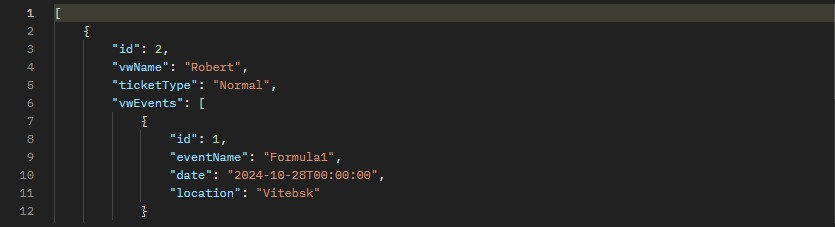
URL: /racingEvent/{id}/viewersByTicket/{ticketType}

**id** and **ticketType** path variable required

Метод: GET

**Ответ:**  
Ok:

Response body:



Status: 200

Error:

Response body:



Status: 404

**2. Получение суммы бюджетов спонсоров определённого ивента:**

URL: /racingEvent/{id}/sponsor-budgets-sum

**id** path variable required

Метод: GET

**Ответ:**  
Ok:

Response body:



Status: 200

Error:

Response body:



Status: 404

**3. Получение спонсоров, которые заключили контракт с ивентом за определённое количество месяцев до начала мероприятия:**

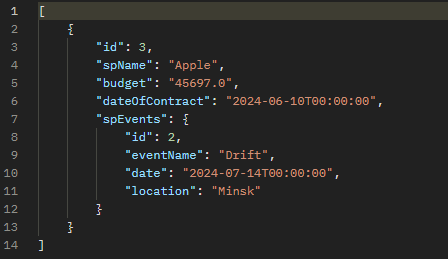
URL: /racingEvent/{id}/sponsorContract/{monthAmount}

**id** and **monthAmount** path variable required

Метод: GET

**Ответ:**  
Ok:

Response body:



Status: 200

Error:

Response body:



Status: 404

Для того чтобы создавать/изменять/удалять сущности, для этого пользователю понадобятся права администратора «ADMIN»

К примеру при попытке создать гонщика выдаст ошибку:

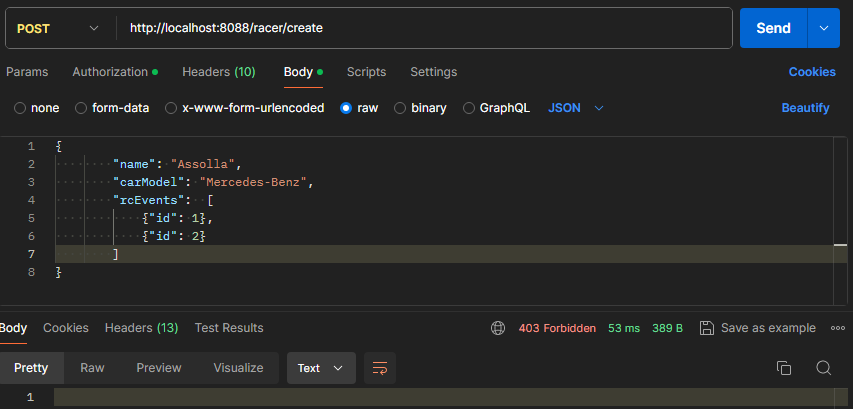


Рис. 17 – Недостаточно прав для создания гонщика

Чтобы это исправить, понадобится получить эти права.

* 1. **Тестирование в роли «ADMIN»**

Для того, чтобы присвоить пользователю права администратора, отправим PUT запрос. Система выдаст права администратора пользователю, на которого мы авторизовались.

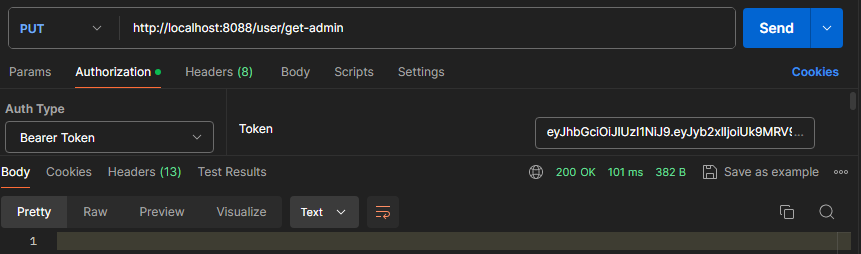


Рис. 18 – Выдача прав администратора пользователю

Теперь с правами администратора нам будут доступны создание/изменение/удаление сущностей.

Приведу примеры работы с сущностями в роли администратора:

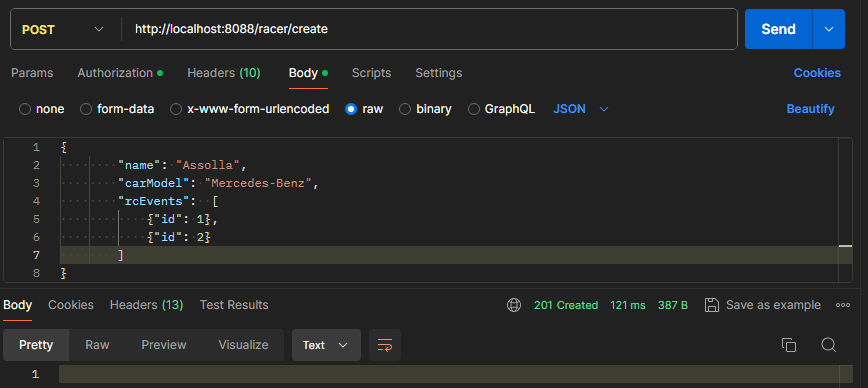


Рис. 19 – Создание гонщика

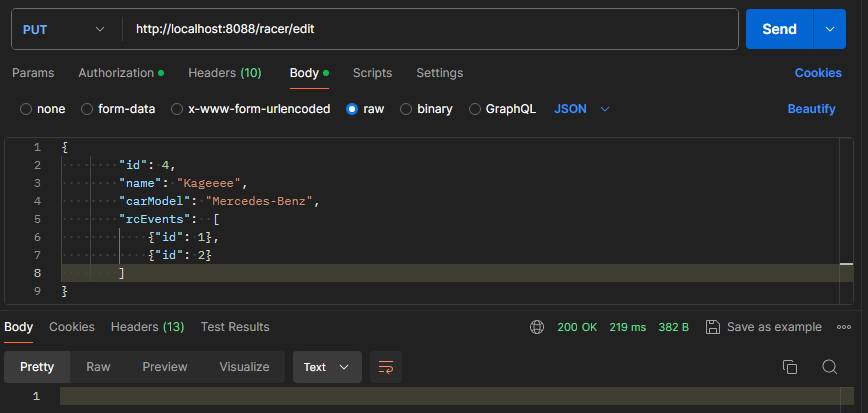


Рис. 20 – Редактирование гонщика

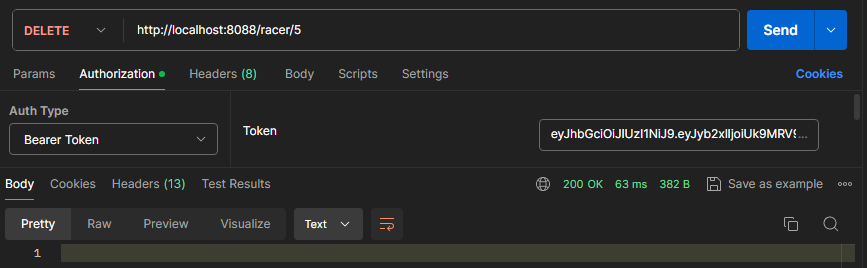


Рис. 21 – Удаление гонщика

**Заключение**

В ходе выполнения работы был разработан REST-сервис “Гоночное мероприятие”. Этот сервис предоставляет возможности просмотра гонщиков, гоночных событий, спонсоров и зрителей для обычных пользователей, а также возможности изменения (создания, удаления) этих данных для админа.

В процессе разработки были использованы современные технологии и инструменты, включая Spring Boot для создания самого сервиса и Postman для его тестирования. Были реализованы функции для создания, чтения, обновления и удаления, а также для фильтрации гоночных событий по различным параметрам, таким как название события.

В работе была реализована довольно простая авторизация. Для авторизации пользователей был использован JWT (JSON Web Token), что позволило обеспечить безопасность и надежность сервиса.

В ходе тестирования сервиса с помощью Postman было установлено, что все функции работают корректно.

В процессе разработки были успешно применены и освоены такие важные аспекты веб-разработки, как работа с базами данных и SQL-запросами, настройка безопасности и авторизации в приложении, а также обработка и устранение ошибок.

Был получен опыт работы с такими технологиями, как Spring Security и JWT для обеспечения безопасности, а также JPA и Hibernate для работы с базой данных. Это позволило глубже понять принципы работы этих технологий и улучшить навыки их применения.

Таким образом, выполнение данной работы позволило не только развить навыки разработки REST-сервисов, но и получить ценный опыт решения реальных задач в области веб-разработки. Этот опыт будет полезен в дальнейшей профессиональной деятельности.

Для ознакомления с моим проектом предлагаю посетить репозиторий на GitHub по ссылке: https://github.com/yujin-bondarev/racingEvent