Министерство образования Республики Беларусь

УО «Витебский государственный технологический университет»

Кафедра «Информационные системы и технологии»

Курсовая работа

по дисциплине «Современные средства разработки серверных приложений»

Разработка REST-сервиса «Учет оборудования»

Выполнил: студент группы 2ИТ-12

Буйвол Иван Павлович

Проверил: доцент, зав. кафедрой ИСиТ

Казаков Вадим Евгеньевич

Витебск, 2024

**СОДЕРЖАНИЕ**

Введение…………………………………………………………………………….

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

2. Описание базы данных, используемой в проекте……………………………..

3. Приложение и его структура …………………………………………………...

4. Тестирование приложения………………………………………………….…...

Заключение………………………………………………………………………….

Приложение………………………………………………………………………….

**Введение**

В современном мире веб-сервисы стали неотъемлемой частью разработки программного обеспечения. Одним из наиболее популярных подходов к созданию веб-сервисов является REST (Representational State Transfer). REST – это архитектурный стиль, который определяет набор ограничений и принципов для построения распределенных приложений. RESTful веб-сервисы используют стандартные протоколы HTTP для обмена данными между клиентом и сервером.

Целью данной курсовой работы является изучение и практическая реализация создания REST-сервиса. В работе будут рассмотрены основные концепции и принципы REST, практические шаги по созданию и использованию REST-сервиса, а также примеры кода на популярных языках программирования. Результатом работы будет готовый пример REST-сервиса, способного обрабатывать HTTP-запросы, взаимодействовать с базой данных и возвращать данные в формате JSON.

В дальнейшем использование REST-сервисов поможет разработчикам создавать масштабируемые и гибкие веб-приложения, способные взаимодействовать с другими системами и сервисами.

Целью данной работы является создание REST-сервера, который бы выполнял набор CRUD команд. Сервер должен создаваться с помощью фреймворка Spring Boot, включать в себя классы на языке Java и компоноваться системой сборки Maven.

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

Стартовой точкой разработки является планирование приложения. В нашем случае REST-сервис «Учет оборудования» должен представлять собой комплексную систему удобного отслеживания действий над оборудованием. Приложение будет предоставлять операции создания, чтения, обновления, удаления, просмотра общей стоимости оборудования и получения полной информации о использующем и самом оборудовании, а также приложение будет реализовывать защищенное подключение, авторизацию и ограничение прав доступа к api сайта в зависимости от роли пользователя.

**Подробное описание ендпоинтов вместе с примерами запросов для них:**

Управление оборудованием:

* Добавление нового оборудования - /api/equipement

пример запроса посредством использования утилиты curl:



* Получения всего списка оборудования - /api/equipment

пример запроса:



* Получение конкретного оборудования по id - /api/equipment/{id}

пример запроса:



* Обновление информации об оборудовании - /api/equipment

пример запроса:



* Удаление конкретного оборудования по id - /api/equipment/{id}

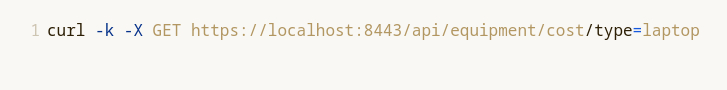
пример запроса:



* Получение общей стоимости об орудования

одного типа – api/equipment/cost/type={type}

пример запроса:



Управление пользователями:

* Добавление нового пользователя - /api/user

пример запроса:



* Получения всего списка пользователей - /api/user

пример запроса:



* Получение конкретного пользователя по id - /api/user/{id}

пример запроса:



* Обновление информации о пользователе - /api/user

пример запроса:



* Удаление конкретного пользователя по id - /api/user/{id}

пример запроса:



Управление историей использования:

* Добавление нового записи - /api/usage-history

пример запроса:



* Получения всей истории - /api/usage-history

пример запроса:



* Получение конкретной записи по id - /api/usage-history/{id}

пример запроса:



* Обновление записи - /api/usage-history

пример запроса:



* Удаление конкретной записи по id - /api/usage-history/{id}

пример запроса:



Описание ролей и прав доступа:

* Не авторизованный пользователь - имеет право

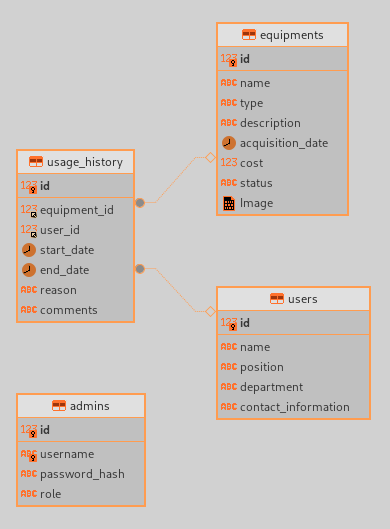
просматривать любые данные

* Manager - имеет все права на регулирование истории использования
* Admin - имеет все права регулирования всех данных

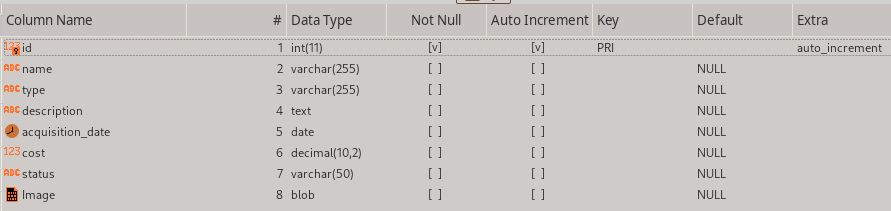
**2. Описание базы данных, используемой в проекте**

Для работы с данными была выбрана MariaDB. MariaDB — это мощная и популярная система управления базами данных, являющаяся форком MySQL. Она была создана разработчиками оригинального MySQL, чтобы обеспечить свободное и открытое будущее для MySQL. MariaDB предоставляет аналогичные функции и совместимость с MySQL, но при этом обладает улучшенной производительностью, расширенной функциональностью и активной поддержкой со стороны сообщества разработчиков.

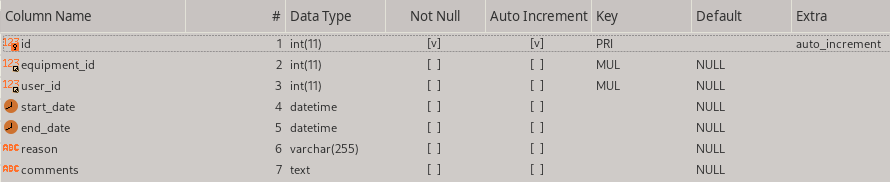
Диаграмма базы данных



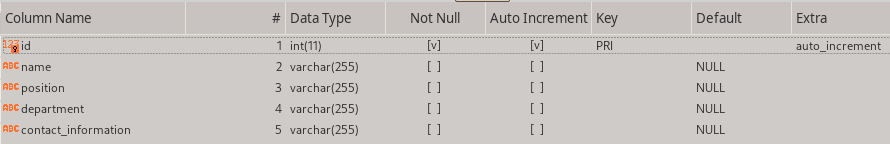
Описание таблицы equipments



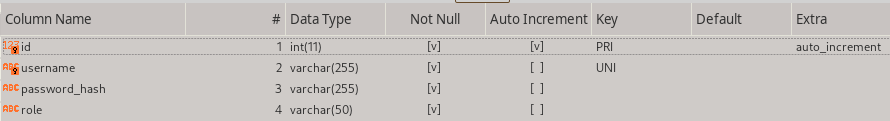
Описание таблицы usage\_history



Описание таблицы users



Описание таблицы admins



Equipments – таблица, в которой хранится информация о оборудовании. В

таблице имеются следующие поля:

* ID – id оборубования
* Name – название
* Type - тип
* Description - описание
* Acquisition Date – дата приобретения
* Cost - цена
* Status - статус
* Image - фото

Users – таблица, в которой хранится информация о пользователях. В

таблице имеются следующие поля:

* ID – id пользователя
* Name - имя
* Position - должность
* Department - департамент
* Contact Information – контактная информация

Usage history – таблица, в которой хранится информация о

использовании оборудовании. В таблице имеются следующие поля:

* ID – id записи
* Equipment ID – id оборудования
* User ID – id пользователя
* Start Date – дата начала использования
* End Date – дата конца использования
* Reason - причина
* Comments – комментарий с доп. информацией

Admins – таблица, в которой хранится информация о администраторах.

В таблице имеются следующие поля:

* ID – id администратора
* Username – id логин
* Password Hash – id хеш пароля
* Role – роль в системе

**3. Приложение и его структура**

Структура директорий и файлов проекта выглядит следующим образом



### 1. Класс Equipment:

* **@Entity и @Table(name = "equipments")**: Класс помечен как сущность JPA и связан с таблицей "equipments" в базе данных.
* **Поля класса соответствуют столбцам таблицы "equipments"**:
  + **id**: Уникальный идентификатор оборудования. Автоматически генерируется при помощи стратегии GenerationType.IDENTITY.
  + **name**: Название оборудования.
  + **type**: Тип оборудования.
  + **description**: Описание оборудования.
  + **acquisitionDate**: Дата приобретения оборудования.
  + **cost**: Стоимость оборудования.
  + **status**: Статус оборудования (например, рабочий, в ремонте).
  + **image**: Ссылка на изображение оборудования.
* **Геттеры и сеттеры для всех полей класса**:
  + getId(): Возвращает идентификатор оборудования.
  + getName() и setName(String name): Возвращает и устанавливает название оборудования.
  + getType() и setType(String type): Возвращает и устанавливает тип оборудования.
  + getDescription() и setDescription(String description): Возвращает и устанавливает описание оборудования.
  + getAcquisitionDate() и setAcquisitionDate(LocalDate acquisitionDate): Возвращает и устанавливает дату приобретения оборудования.
  + getCost() и setCost(double cost): Возвращает и устанавливает стоимость оборудования.
  + getStatus() и setStatus(String status): Возвращает и устанавливает статус оборудования.
  + getImage() и setImage(String image): Возвращает и устанавливает ссылку на изображение оборудования.

### 2. Класс User:

* **@Entity и @Table(name = "users")**: Класс помечен как сущность JPA и связан с таблицей "users" в базе данных.
* **Поля класса соответствуют столбцам таблицы "users"**:
  + **id**: Уникальный идентификатор пользователя. Автоматически генерируется при помощи стратегии GenerationType.IDENTITY.
  + **name**: Имя пользователя.
  + **position**: Должность пользователя.
  + **department**: Отдел, в котором работает пользователь.
  + **contactInformation**: Контактная информация пользователя.
* **Геттеры и сеттеры для всех полей класса**:
  + getId(): Возвращает идентификатор пользователя.
  + getName() и setName(String name): Возвращает и устанавливает имя пользователя.
  + getPosition() и setPosition(String position): Возвращает и устанавливает должность пользователя.
  + getDepartment() и setDepartment(String department): Возвращает и устанавливает отдел пользователя.
  + getContactInformation() и setContactInformation(String contactInformation): Возвращает и устанавливает контактную информацию пользователя.

### 3. Класс UsageHistory:

* **@Entity и @Table(name = "usage\_history")**: Класс помечен как сущность JPA и связан с таблицей "usage\_history" в базе данных.
* **Поля класса соответствуют столбцам таблицы "usage\_history"**:
  + **id**: Уникальный идентификатор истории использования. Автоматически генерируется при помощи стратегии GenerationType.IDENTITY.
  + **equipment**: Ссылка на оборудование, использованное в истории.
  + **user**: Ссылка на пользователя, связанного с историей использования.
  + **startDate**: Дата начала использования оборудования.
  + **endDate**: Дата окончания использования оборудования.
  + **reason**: Причина использования оборудования.
  + **comments**: Комментарии по использованию оборудования.
* **Геттеры и сеттеры для всех полей класса**:
  + getId(): Возвращает идентификатор истории использования.
  + getEquipment() и setEquipment(Equipment equipment): Возвращает и устанавливает оборудование, использованное в истории.
  + getUser() и setUser(User user): Возвращает и устанавливает пользователя, связанного с историей использования.
  + getStartDate() и setStartDate(LocalDate startDate): Возвращает и устанавливает дату начала использования оборудования.
  + getEndDate() и setEndDate(LocalDate endDate): Возвращает и устанавливает дату окончания использования оборудования.
  + getReason() и setReason(String reason): Возвращает и устанавливает причину использования оборудования.
  + getComments() и setComments(String comments): Возвращает и устанавливает комментарии по использованию оборудования.

### 4. Класс Admin:

* **@Entity и @Table(name = "admins")**: Класс помечен как сущность JPA и связан с таблицей "admins" в базе данных.
* **Поля класса соответствуют столбцам таблицы "admins"**:
  + **id**: Уникальный идентификатор администратора. Автоматически генерируется при помощи стратегии GenerationType.IDENTITY.
  + **username**: Имя пользователя администратора. Поле не может быть пустым и должно быть уникальным.
  + **passwordHash**: Хеш пароля администратора. Поле не может быть пустым.
  + **role**: Роль администратора (например, "SUPER\_ADMIN", "ADMIN"). Поле не может быть пустым.
* **Геттеры и сеттеры для всех полей класса**:
  + getId(): Возвращает идентификатор администратора.
  + getUsername() и setUsername(String username): Возвращает и устанавливает имя пользователя администратора.
  + getPasswordHash() и setPasswordHash(String passwordHash): Возвращает и устанавливает хеш пароля администратора.
  + getRole() и setRole(String role): Возвращает и устанавливает роль администратора.

### Описание контроллера BaseController

Контроллер BaseController является абстрактным классом, который предоставляет основные CRUD операции для сущностей. Этот контроллер может быть расширен другими контроллерами для работы с конкретными типами сущностей. Он использует JpaRepository для взаимодействия с базой данных и аннотации Spring MVC для обработки HTTP запросов.

#### Основные методы и их описание:

1. **Создание сущности (POST)**:
   * **Метод**: @PostMapping
   * **Описание**: Создает новую сущность и сохраняет её в базе данных.
   * **URL**: /api
   * **Входные данные**: JSON представление сущности.
   * **Выходные данные**: Созданная сущность с кодом статуса 201 Created.
2. **Получение всех сущностей (GET)**:
   * **Метод**: @GetMapping
   * **Описание**: Возвращает список всех сущностей из базы данных.
   * **URL**: /api
   * **Выходные данные**: Список всех сущностей.
3. **Получение сущности по ID (GET)**:
   * **Метод**: @GetMapping("/{id}")
   * **Описание**: Возвращает сущность по её идентификатору.
   * **URL**: /api/{id}
   * **Входные данные**: Идентификатор сущности.
   * **Выходные данные**: Сущность, если она существует, или код статуса 404 Not Found, если не существует.
4. **Обновление сущности (PUT)**:
   * **Метод**: @PutMapping("/{id}")
   * **Описание**: Обновляет существующую сущность по её идентификатору.
   * **URL**: /api/{id}
   * **Входные данные**: Идентификатор сущности и JSON представление обновленной сущности.
   * **Выходные данные**: Обновленная сущность, если она существует, или код статуса 404 Not Found, если не существует.
5. **Удаление сущности (DELETE)**:
   * **Метод**: @DeleteMapping("/{id}")
   * **Описание**: Удаляет сущность по её идентификатору.
   * **URL**: /api/{id}
   * **Входные данные**: Идентификатор сущности.
   * **Выходные данные**: Код статуса 200 OK, если сущность успешно удалена, или код статуса 404 Not Found, если сущность не найдена.

### Основные особенности:

* **@RequestMapping("/api")**: Задает базовый URL для всех методов контроллера.
* **Абстрактный класс**: Этот класс не может быть инстанциирован напрямую. Он должен быть расширен конкретными контроллерами, которые будут указывать тип сущности и идентификатора.
* **Использование** JpaRepository: Для выполнения операций с базой данных используется репозиторий JPA.
* **HTTP статус-коды**: Методы возвращают соответствующие HTTP статус-коды (201 Created, 200 OK, 404 Not Found) для указания результата операции.

### Описание контроллера EquipmentController

Контроллер EquipmentController расширяет базовый контроллер BaseController и предоставляет дополнительные методы для работы с сущностями типа Equipment. Он также взаимодействует с репозиториями EquipmentRepository и UsageHistoryRepository.

#### Основные методы и их описание:

1. **Создание сущности (POST)**:
   * **Метод**: @PostMapping
   * **Описание**: Наследуется от BaseController. Создает новую сущность Equipment и сохраняет её в базе данных.
   * **URL**: /api/equipment
   * **Входные данные**: JSON представление сущности Equipment.
   * **Выходные данные**: Созданная сущность с кодом статуса 201 Created.
2. **Получение всех сущностей (GET)**:
   * **Метод**: @GetMapping
   * **Описание**: Наследуется от BaseController. Возвращает список всех сущностей Equipment из базы данных.
   * **URL**: /api/equipment
   * **Выходные данные**: Список всех сущностей Equipment.
3. **Получение сущности по ID (GET)**:
   * **Метод**: @GetMapping("/{id}")
   * **Описание**: Наследуется от BaseController. Возвращает сущность Equipment по её идентификатору.
   * **URL**: /api/equipment/{id}
   * **Входные данные**: Идентификатор сущности.
   * **Выходные данные**: Сущность Equipment, если она существует, или код статуса 404 Not Found, если не существует.
4. **Обновление сущности (PUT)**:
   * **Метод**: @PutMapping("/{id}")
   * **Описание**: Наследуется от BaseController. Обновляет существующую сущность Equipment по её идентификатору.
   * **URL**: /api/equipment/{id}
   * **Входные данные**: Идентификатор сущности и JSON представление обновленной сущности Equipment.
   * **Выходные данные**: Обновленная сущность Equipment, если она существует, или код статуса 404 Not Found, если не существует.
5. **Удаление сущности (DELETE)**:
   * **Метод**: @DeleteMapping("/{id}")
   * **Описание**: Переопределяет метод из BaseController. Удаляет сущность Equipment по её идентификатору, если она не используется в истории использования.
   * **URL**: /api/equipment/{id}
   * **Входные данные**: Идентификатор сущности.
   * **Выходные данные**: Код статуса 200 OK, если сущность успешно удалена, или код статуса 404 Not Found, если сущность не найдена. Возвращает код статуса 409 Conflict, если сущность используется в истории использования.
6. **Получение общей стоимости оборудования по типу (GET)**:
   * **Метод**: @GetMapping("/cost/type={type}")
   * **Описание**: Возвращает общую стоимость оборудования по указанному типу.
   * **URL**: /api/equipment/cost/type={type}
   * **Входные данные**: Тип оборудования в качестве параметра запроса.
   * **Выходные данные**: Общая стоимость оборудования указанного типа.

### Основные особенности:

* **@RestController и @RequestMapping("api/equipment")**: Аннотации указывают, что этот класс является контроллером REST и что все его методы будут обрабатывать запросы, начинающиеся с /api/equipment.
* **Конструктор с @Autowired**: Конструктор инициализирует EquipmentRepository и UsageHistoryRepository для взаимодействия с базой данных.
* **Расширение** BaseController: Наследование основных CRUD операций от базового контроллера и добавление специфичных для оборудования методов.
* **Проверка перед удалением**: Перед удалением оборудования проверяется, не используется ли оно в истории использования, чтобы предотвратить удаление данных, которые могут быть необходимы.

### Описание контроллера UserController

Контроллер UserController расширяет базовый контроллер BaseController и предоставляет дополнительные методы для работы с сущностями типа User. Он также взаимодействует с репозиториями UserRepository и UsageHistoryRepository.

#### Основные методы и их описание:

1. **Создание сущности (POST)**:
   * **Метод**: @PostMapping
   * **Описание**: Наследуется от BaseController. Создает новую сущность User и сохраняет её в базе данных.
   * **URL**: /api/user
   * **Входные данные**: JSON представление сущности User.
   * **Выходные данные**: Созданная сущность User с кодом статуса 201 Created.
2. **Получение всех сущностей (GET)**:
   * **Метод**: @GetMapping
   * **Описание**: Наследуется от BaseController. Возвращает список всех сущностей User из базы данных.
   * **URL**: /api/user
   * **Выходные данные**: Список всех сущностей User.
3. **Получение сущности по ID (GET)**:
   * **Метод**: @GetMapping("/{id}")
   * **Описание**: Наследуется от BaseController. Возвращает сущность User по её идентификатору.
   * **URL**: /api/user/{id}
   * **Входные данные**: Идентификатор сущности.
   * **Выходные данные**: Сущность User, если она существует, или код статуса 404 Not Found, если не существует.
4. **Обновление сущности (PUT)**:
   * **Метод**: @PutMapping("/{id}")
   * **Описание**: Наследуется от BaseController. Обновляет существующую сущность User по её идентификатору.
   * **URL**: /api/user/{id}
   * **Входные данные**: Идентификатор сущности и JSON представление обновленной сущности User.
   * **Выходные данные**: Обновленная сущность User, если она существует, или код статуса 404 Not Found, если не существует.
5. **Удаление сущности (DELETE)**:
   * **Метод**: @DeleteMapping("/{id}")
   * **Описание**: Переопределяет метод из BaseController. Удаляет сущность User по её идентификатору, если она не имеет истории использования.
   * **URL**: /api/user/{id}
   * **Входные данные**: Идентификатор сущности.
   * **Выходные данные**: Код статуса 200 OK, если сущность успешно удалена, или код статуса 404 Not Found, если сущность не найдена. Возвращает код статуса 409 Conflict, если сущность имеет историю использования.

### Основные особенности:

* **@RestController и @RequestMapping("/api/user")**: Аннотации указывают, что этот класс является контроллером REST и что все его методы будут обрабатывать запросы, начинающиеся с /api/user.
* **Конструктор с @Autowired**: Конструктор инициализирует UserRepository и UsageHistoryRepository для взаимодействия с базой данных.
* **Расширение** BaseController: Наследование основных CRUD операций от базового контроллера и добавление специфичных для пользователей методов.
* **Проверка перед удалением**: Перед удалением пользователя проверяется, имеет ли он историю использования, чтобы предотвратить удаление данных, которые могут быть необходимы.

### Описание контроллера UsageHistoryController

Контроллер UsageHistoryController расширяет базовый контроллер BaseController и предоставляет дополнительные методы для работы с сущностями типа UsageHistory. Он также взаимодействует с репозиториями UsageHistoryRepository, EquipmentRepository и UserRepository.

#### Основные методы и их описание:

1. **Создание сущности (POST)**:
   * **Метод**: @PostMapping
   * **Описание**: Наследуется от BaseController. Создает новую сущность UsageHistory и сохраняет её в базе данных.
   * **URL**: /api/usage-history
   * **Входные данные**: JSON представление сущности UsageHistory.
   * **Выходные данные**: Созданная сущность UsageHistory с кодом статуса 201 Created.
2. **Получение всех сущностей (GET)**:
   * **Метод**: @GetMapping
   * **Описание**: Наследуется от BaseController. Возвращает список всех сущностей UsageHistory из базы данных.
   * **URL**: /api/usage-history
   * **Выходные данные**: Список всех сущностей UsageHistory.
3. **Получение сущности по ID (GET)**:
   * **Метод**: @GetMapping("/{id}")
   * **Описание**: Наследуется от BaseController. Возвращает сущность UsageHistory по её идентификатору.
   * **URL**: /api/usage-history/{id}
   * **Входные данные**: Идентификатор сущности.
   * **Выходные данные**: Сущность UsageHistory, если она существует, или код статуса 404 Not Found, если не существует.
4. **Обновление сущности (PUT)**:
   * **Метод**: @PutMapping("/{id}")
   * **Описание**: Переопределяет метод из BaseController. Обновляет существующую сущность UsageHistory по её идентификатору.
   * **URL**: /api/usage-history/{id}
   * **Входные данные**: Идентификатор сущности и JSON представление обновленной сущности UsageHistory.
   * **Выходные данные**: Обновленная сущность UsageHistory, если она существует, или код статуса 404 Not Found, если не существует.

#### Основные особенности:

* **@RestController и @RequestMapping("/api/usage-history")**: Аннотации указывают, что этот класс является контроллером REST и что все его методы будут обрабатывать запросы, начинающиеся с /api/usage-history.
* **Конструктор с @Autowired**: Конструктор инициализирует UsageHistoryRepository, EquipmentRepository и UserRepository для взаимодействия с базой данных.
* **Расширение** BaseController: Наследование основных CRUD операций от базового контроллера и переопределение метода update для специфичного обновления истории использования.
* **Проверка перед обновлением**: Перед обновлением истории использования проверяется существование оборудования и пользователя, а также корректность временных меток и причины.

### Описание класса конфигурации WebSecurityConfig

Класс WebSecurityConfig является конфигурационным классом для настройки безопасности веб-приложения с использованием Spring Security. Он аннотирован @Configuration, что позволяет Spring использовать его для настройки контекста приложения. Также он аннотирован @EnableWebSecurity, чтобы включить поддержку безопасности веб-приложения.

#### Основные методы и их описание:

1. **configure(HttpSecurity http)**:
   * **Описание**: Этот метод настраивает правила безопасности для различных HTTP запросов.
   * **CSRF**: Отключает защиту от CSRF атак.
   * **Авторизация**: Устанавливает правила доступа для различных URL-адресов:
     + Для всех HTTP GET запросов на URL, начинающиеся с /api/, доступ разрешен для всех пользователей.
     + Для всех HTTP POST, PUT и DELETE запросов на URL, начинающиеся с /api/, доступ разрешен только для пользователей с ролью "ADMIN".
     + Для HTTP POST, PUT и DELETE запросов на /api/usage-history, доступ разрешен только для пользователей с ролью "MANAGER".
     + Для всех остальных запросов требуется аутентификация пользователя.
   * **HTTP Basic Authentication**: Включает базовую аутентификацию HTTP.
2. **configure(AuthenticationManagerBuilder auth)**:
   * **Описание**: Этот метод настраивает менеджер аутентификации.
   * **UserDetailsService**: Устанавливает CustomUserDetailsService в качестве сервиса пользователей для аутентификации.
   * **PasswordEncoder**: Устанавливает BCryptPasswordEncoder в качестве кодировщика паролей.
3. **passwordEncoder()**:
   * **Описание**: Создает и возвращает экземпляр BCryptPasswordEncoder для кодирования паролей.
4. **authenticationManagerBean()**:
   * **Описание**: Создает и возвращает экземпляр AuthenticationManager, который используется для аутентификации пользователей в приложении.

#### Основные особенности:

* **@Configuration и @EnableWebSecurity**: Аннотации, указывающие, что этот класс является конфигурационным классом для Spring и должен включать поддержку безопасности веб-приложения.
* **WebSecurityConfigurerAdapter**: Расширение этого класса позволяет определить пользовательские настройки безопасности для веб-приложения.
* **Авторизация на основе ролей**: Устанавливаются правила доступа для различных URL-адресов на основе ролей пользователей.
* **PasswordEncoder**: Используется для безопасного хранения паролей пользователей с использованием хэш-функции.
* **AuthenticationManager**: Предоставляет возможность аутентификации пользователей в приложении.

**4. Тестирование приложения**

Протестируем приложение путем отправки выше приведенных запросов

Управление оборудованием:

* Получения всего списка оборудования - /api/equipment



* Получение конкретного оборудования по id - /api/equipment/{id}



* Добавление нового оборудования - /api/equipement



* Обновление информации об оборудовании - /api/equipment

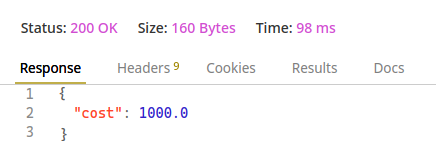


* Удаление конкретного оборудования по id - /api/equipment/{id}



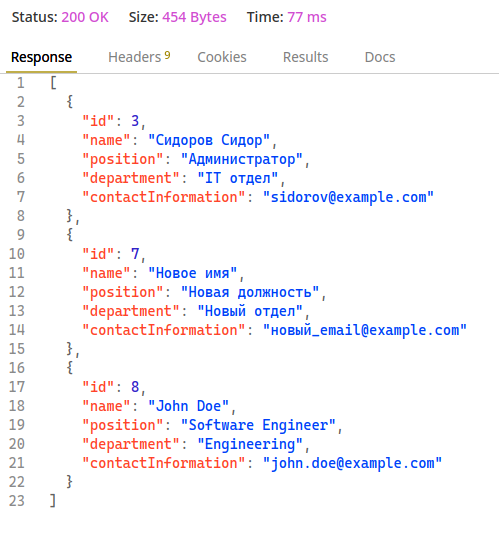
* Получение общей стоимости оборудования

одного типа – api/equipment/cost/type={type}

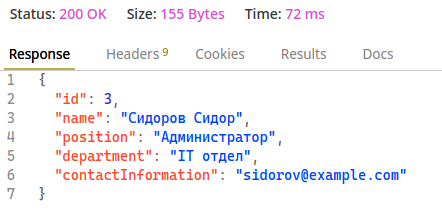


Управление пользователями:

* Получения всего списка пользователей - /api/user



* Получение конкретного пользователя по id - /api/user/{id}



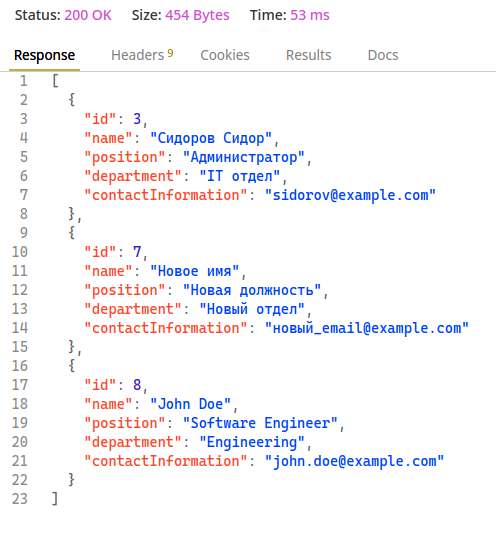
* Добавление нового пользователя - /api/user



* Обновление информации о пользователе - /api/user



* Удаление конкретного пользователя по id - /api/user/{id}

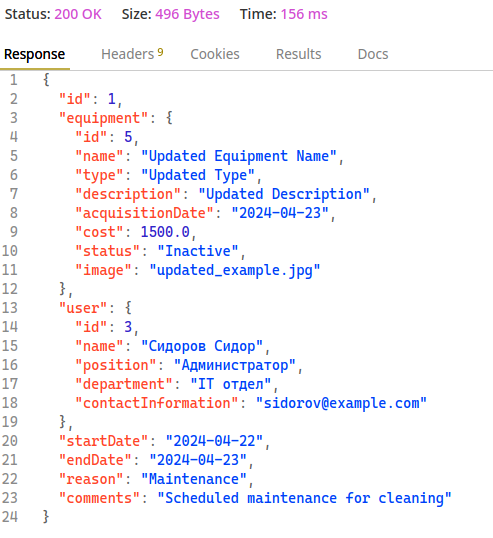


Управление историей использования:

* Получения всей истории - /api/usage-history



* Получение конкретной записи по id - /api/usage-history/{id}



* Добавление нового записи - /api/usage-history
* Обновление записи - /api/usage-history



* Удаление конкретной записи по id - /api/usage-history/{id}



**Заключение**

По ходу нашего общения был разработан REST-сервис учета оборудования, предоставляющий следующий функционал:

1. Управление оборудованием:
   * Реализована возможность просмотра всех устройств и устройств по идентификатору.
   * Добавлена функция создания и удаления оборудования.
   * Разработана функция обновления информации о каждом устройстве.
   * Предусмотрено отображение всех устройств и дополнительной информации о каждом из них.
2. Управление историей использования:
   * Добавлена возможность просмотра всех записей истории использования и записей по идентификатору.
   * Реализована функция создания и удаления записей истории использования.
   * Разработана функция обновления информации о каждой записи истории использования.
   * Предусмотрено отображение всех записей истории использования и связанных с ними данных.
3. Управление пользователями:
   * Реализованы методы для получения всех пользователей и пользователей по идентификатору.
   * Добавлена функция создания и удаления пользователей.
   * Предусмотрено обновление информации о пользователях и их ролях.

В результате тестирования была подтверждена корректность работы всех функций сервиса. В процессе разработки были успешно применены и освоены такие важные аспекты веб-разработки, как работа с базами данных, настройка безопасности и авторизации в приложении, а также обработка и устранение ошибок.

Cервис учета оборудования обеспечивает удобство пользователей и эффективное взаимодействие с информацией о оборудовании в онлайн среде.

**Приложение**

Для ознакомления с проектом можно посетить GitHub репозиторий по ссылке:

[](https://github.com/ndgde/Equipment-Tracker)https://github.com/ndgde/Equipment-Tracker