МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. Шухова»

(БГТУ им. В. Г. Шухова)

Курсовая работа

Дисциплина «Интерфейсы вычислительных систем»

На тему: «Реализация REST-приложения с использованием Spring Framework»

Выполнил: студент группы ВТ-41

Ковалёв И. Д.

Проверил: Торопчин Д. А.

г. Белгород

2020 г.

**Оглавление**

[**Введение** 3](#_Toc59557681)

[**Глава 1.** **Теоретические сведения** 4](#_Toc59557682)

[**1.1 REST архитектура** 4](#_Toc59557683)

[**1.2 Spring Framework** 5](#_Toc59557684)

[**Глава 2. Разработка приложения** 9](#_Toc59557685)

[**Глава 3. Тестирование ПО** 13](#_Toc59557686)

[**Скриншоты** 17](#_Toc59557687)

[**Заключение** 18](#_Toc59557688)

[**Список литературы** 19](#_Toc59557689)

[**Приложения** 20](#_Toc59557690)

# **Введение**

На сегодняшний день сервисы, построенные на основе архитектуры REST являются наиболее часто используемые, особенно если речь идет о разработке веб-приложений. Такая архитектурная модель является довольно простой в реализации, и предлагает большой набор основополагающих принципов, которые позволяют реализовывать высокопроизводительные и масштабируемые программные сервисы. Термин «REST» был введен Роем Филдингом в 2000 году в своей диссертации «Архитектурные стили и дизайн сетевых программных архитектур», где он подвел теоретическую основу под способ взаимодействия клиентов и серверов во Всемирной паутине, абстрагировав его и назвав «Передачей представительного состояния», а также описал концепцию построения распределенного приложения, при которой каждый запрос клиента к серверу содержит в себе исчерпывающую информацию о желаемом ответе сервера, при этом сервер не обязан сохранять информацию о состоянии клиента[1].

# **Глава 1.** **Теоретические сведения**

## **1.1 REST архитектура**

REST представляет собой стиль архитектуры программного обеспечения, который, в том числе, используется для построения распределенных масштабируемых веб-сервисов.

Следует заметить, что веб-сервис, построенный согласно принципам REST, будет иметь такие свойства:

Применение данной архитектуры привнесет в разрабатываемую систему такие положительные характеристики, как:

1. Унифицированность интерфейса,
2. Гибкость компонентов для внесения в них изменений и отсутствие необходимости перестройки архитектуры приложения для внесения незначительных правок;
3. Прозрачность связей между компонентами системы
4. Переносимость компонентов системы путем перемещения программного кода вместе с данными;
5. Отказоустойчивость системы в целом, даже при условии, что некоторые из компонентов вышли из строя.

Однако, следует учитывать, что существуют обязательные ограничения для построения распределенных приложений. Данные ограничения определяют работу сервера в рамках обработки и ответов на запросы клиентов. Придерживаясь этих ограничений, система обретает такие необходимые свойства, как производительность, масштабируемость, простота, гибкость, переносимость и надежность.

К обязательным ограничениям относятся:

1. Приведение архитектуры приложения к модели клиент-сервер. Разделение интерфейса клиента от серверной части, которая хранит данные, повышает переносимость кода на другие платформы. В свою очередь, упрощение серверной части заметно повышает такую характеристику системы, как масштабируемость.
2. Сервер не должен хранить состояние клиента в период между его запросами. Без исключения все запросы клиента должны быть составлены таким образом, чтобы сервер получил всю необходимую ему информацию для выполнения запроса. При этом состояние сессии сохраняется на стороне клиента, и может быть передано в любой другой сервис.
3. Клиенты и промежуточные узлы могут выполнять кэширование ответов сервера. Грамотное использование кэширования способно почти полностью устранить некоторые взаимодействия между сервером и клиентом, таким образом повышая производительность системы.
4. Унификация интерфейса, а именно выполнение таких требований:
   1. Все ресурсы идентифицируются в запросах, при этом ресурсы концептуально отделены от представлений, которые получают клиенты.
   2. В случае, если клиент хранит представление ресурса – он обладает достаточной информацией для модификации или удаления ресурса.
   3. Каждое сообщение содержит достаточно информации, чтобы понимать, каким образом его обрабатывать.
   4. Клиенты могут изменять состояние системы только через те действия, которые были определены гипермедиа на сервере.
5. Клиент, как правило, не способен точно определить, с чем он взаимодействует: напрямую с сервером или с некоторым промежуточным узлом. Применение промежуточных серверов повышает масштабируемость и отказоустойчивость системы в целом за счет балансировки нагрузки и распределенного кэширования.
6. Расширение функциональности клиента за счет загрузки кода с сервера в виде апплетов или сценариев. Следует заметить, что данное требование является необязательным.

## **1.2 Spring Framework**

Разрабатываемое приложение будет строиться на REST архитектуре, используя Spring Framework.

Spring Framework – универсальный фреймворк с открытым исходным кодом для Java-платформы[2]. Данный фреймворк дают большую свободу разработчику в проектировании, и представляет хорошо документированные и легкие в использовании средства решения различных проблем, которые возникают при создании приложений корпоративного масштаба. При всем этом ядро Spring является достаточно гибким и расширяемым, благодаря чему и приобрело большую популярность среди java-разработчиков.

Spring Framework содержит большое количество модулей, которые делятся на структурные элементы, среди которых:

1. Inversion of Control-контейнер: конфигурирование компонентов приложений и управление жизненным циклом Java-объектов.
2. Фреймворк доступа к данным: работает с системами управления реляционными базами данных на Java-платформе, используя JDBC- и ORM-средства и обеспечивая решения задач, которые повторяются в большом числе Java-based environments.
3. Фреймворк управления транзакциями: координация различных API управления транзакциями и инструментарий настраиваемого управления транзакциями для объектов Java.
4. Фреймворк MVC: каркас, основанный на HTTP и сервлетах, предоставляющий множество возможностей для расширения и настройки (customization).
5. Фреймворк удалённого доступа: конфигурируемая передача Java-объектов через сеть в стиле RPC, поддерживающая RMI, CORBA, HTTP-based протоколы, включая web-сервисы (SOAP).
6. Фреймворк аутентификации и авторизации: конфигурируемый инструментарий процессов аутентификации и авторизации, поддерживающий много популярных и ставших индустриальными стандартами протоколов, инструментов, практик через дочерний проект Spring Security (ранее известный как Acegi).
7. Тестирование: каркас, поддерживающий классы для написания модульных и интеграционных тестов. [2]

Поскольку целью данной работы является создание backend`а для веб приложения, более подробно рассмотрим Spring Web MVC.

Данный фреймворк обеспечивает архитектуру паттерна Модель – отображение – контроллер.

Модели реализуют инкапсуляцию данных приложения, здесь представлены Java-объектами.

Отображения, в свою очередь, отвечают за отображение данных моделей.

Контроллер призван обрабатывать запрос пользователя, создавать соответствующую модель и передавать ее для отображения в вид.

В целом, реализация REST API приложения состоит из таких этапов:

1. Анализ предметной области, выделение основных сущностей и установка связей между ними.
2. Реализация моделей.
3. Реализация контроллеров.
4. Реализация отображения.

В целом, создание моделей является довольно простым процессом. Для этого необходимо описать класс, который будет отображать хранимую структуру данных.

Приведем пример. Допустим, необходимо создать модель, которая отображает хранимые данные для различных чашек. Код будет выглядеть так:

@Getter  
@Setter  
@EqualsAndHashCode  
@ToString  
@Entity  
@AllArgsConstructor  
public class Cup {  
 @Id  
 @GeneratedValue(strategy = GenerationType.AUTO)  
 private Long id;  
 private String name;  
 private String color;  
}

Следует заметить, что использование Project Lombok значительно уменьшает количество кода. В вышеуказанном примере благодаря этому удалось сократить код до описания полей класса, при этом все геттеры, сеттеры, конструкторы будут созданы аннотациями @Getter, @Setter, @AllArgsConstructor.

Следующим этапом является создание репозитория. Репозиторий – это несколько интерфейсов, которые используют модели для взаимодействия с ними.

В нашем случае, репозиторий будет выглядеть так:

@Repository  
public interface CupRepository extends JpaRepository<Cup, Long> {  
 List<Cup> findCupByName(String name);  
}

Далее описывается контроллер, который должен содержать экземпляр репозитория. Контроллер позволяет установить, по каким url будут отдаваться или приниматься данные со стороны клиента. Для этого используются аннотации @GetMapping, @PostMapping и так далее.

В рамках примера с чашками контроллер будет иметь такой вид:

@RestController

public class CupController {

private final CupRepository repository;

CupController(CupRepository repository) {

this.repository = repository;

}

@CrossOrigin(origins = "\*")

@GetMapping("/Cups")

List<Cup> all() {

return repository.findAll();

}

@CrossOrigin(origins = "\*")

@PostMapping("/Cups")

Cup newCup(@RequestBody Cup newCup){

return repository.save(newCup);

}

@CrossOrigin(origins = "\*")

@GetMapping("/Cups/{id}")

Optional<Cup> one(@PathVariable Long id){

return repository.findById(id);

}

}

Здесь используется аннотация @CrossOrigin. Она позволяет конфигурировать CORS-заголовки, что предоставляет возможность доступа сервисами из другого домена.

Каждый из методов контроллера конфигурирует URL, по которому будет происходить взаимодействие клиента с сервером, и http-метод взаимодействия, в данном примере – GET и POST. Очевидно, что данный вариант не является единственным вариантом реализации контроллера, однако такой способ является простым и позволяет программисту наглядно осознавать все возможности реализуемой API с минимумом «подкапотной магии».

# **Глава 2. Разработка приложения**

Очевидно, что процесс разработки приложения следует разделить по этапам, первым из которых будет анализ предметной области, выделение сущностей и связей между ними, и составление диаграммы классов.

В качестве предметной области был выбран сервис для поиска музыки.

Однозначно можно сказать, что к сущностям данной предметной области можно отнести:

1. Исполнитель, информация о нем должна содержать:
   * Название
   * Описание
   * Дата создания/рождения
   * Рейтинг
   * Логотип
2. Альбом
   * Название
   * Дата выхода
   * Обложка
   * Рейтинг
3. Песня
   * Название
   * Длительность
   * Рейтинг
4. Жанр
   * Название

При этом связи между этими сущностями можно определить таким образом:

У одного исполнителя может быть несколько альбомов, в свою очередь альбом может содержать несколько песен. Сам альбом может быть отнесен к нескольким жанрам.

Составим схему «Сущность-связь»:



Реализуем данную схему. Создадим модели Band, Track, Genre и Album.

Исходный код модели Album:

@Getter  
@Setter  
@EqualsAndHashCode  
@ToString  
@Entity  
@AllArgsConstructor  
public class Album {  
 @Id  
 @GeneratedValue(strategy = GenerationType.AUTO)  
 @Column(name = "album\_id")  
 private Long id;  
 private String title;  
 private LocalDate release\_date;  
 private String cover;  
 private Integer rating;  
 @ManyToOne  
 @JoinColumn(name = "band", referencedColumnName = "id")  
 private Band band;  
  
 @ManyToMany  
 @JoinTable(  
 name = "album\_genres",  
 joinColumns = @JoinColumn(name="album\_id"),  
 inverseJoinColumns = @JoinColumn(name="genre\_id")  
 )  
 private List<Genre> genres;  
  
}

Рассмотрим данную модель. В качестве primary key используется поле Long id с аннотациями @Id, @GeneratedValue, @Column. В частности, применение второй аннотации позволяет совершать POST-запросы к серверу, не вкладывая в JSON поле id, ведь оно будет сгенерировано автоматически.

Связи один ко многим и многие ко многим реализуются с помощью аннотаций @OneToMany и @ManyToMany соответственно. В данном случае объекта класса Band является контейнером, который указывает на исполнителя, которому принадлежит альбом. С помощью аргументов name и referencedColumnName мы указываем, какое поле будет использоваться в качетсве foreign key.

Со связью многие ко многим ситуация немного сложнее: здесь необходимо условно описать промежуточную таблицу для хранения отношений между двумя объектами. В данном случае описание содержит название таблицы, столбцы, содержащий внешние ключи альбомов и жанров.

Исходный код остальных моделей приведен в приложении 2-4 соответственно.

Далее были созданы репозитории для каждого из классов. Ниже приведен исходный код для репозитория альбомов:

@Repository  
public interface AlbumRepository extends JpaRepository<Album, Long> {  
  
 List<Album> findAlbumByTitle(String name);  
  
}

Здесь была описана сигнатура метода findAlbumByTitle, которая возвращает объект(ы) класса Album с названием name. На этом моменте следует поблагодарить разработчиков Spring Framework – реализация данного метода создается автоматически, что заметно экономит время, особенно в случаях, когда таких методов надо сделать несколько и для нескольких классов.

Далее опишем контроллеры. В качестве примера возьмем BandController:

@RestController  
public class BandController {  
  
 private final BandRepository repository;  
  
 BandController(BandRepository repository) {  
 this.repository = repository;  
 }  
  
 @CrossOrigin(origins = "\*")  
 @GetMapping("/bands")  
 List<Band> all() {  
 return (List<Band>) repository.findAll();  
 }  
  
 @CrossOrigin(origins = "\*")  
 @PostMapping("/bands")  
 Band newBand(@RequestBody Band newBand) {  
 return repository.save(newBand);  
 }  
  
 @CrossOrigin(origins = "\*")  
 @GetMapping("/bands/{id}")  
 Optional<Band> one(@PathVariable Long id){  
 return repository.findById(id);  
 }  
  
}

По результатам, для реализуемого API мы получаем URL hostname:8080/bands.

Были описаны методы GET и POST, при этом благодаря методу newBand клиент может получать JSON, содержащий данные об одном экземпляре класса Band, primary key которого был указан в URL.

Остальные контроллеры принципиально ничем не отличаются от приведенного выше, кроме контроллера AlbumController. Здесь был реализован поиск:

@CrossOrigin(origins = "\*")  
@GetMapping("/albums")  
List<Album> all(@RequestParam(required = false) String title) {  
 if (title == null) {  
 return repository.findAll();  
 }  
 return repository.findAlbumByTitle(title);  
}

Здесь и пригодился метод findAlbumByTitle. В качестве параметров запроса был описан параметр title, который является необязательным. Теперь, чтобы совершить поиск по названию, достаточно воспользоваться URL hostname:8080/bands/?title= «Некоторое название».

# **Глава 3. Тестирование ПО**

Проведем тестирование реализованного API с помощью приложения Advanced REST Client. Попробуем добавить некоторые тестовые данные, чтобы выявить неполадки, если такие присутствуют.

Был выбран такой набор тестовых данных:

Для <http://localhost:8080/bands/>:

{

"name": "Band with no name",

"description": "And no decription",

"created": "2020-12-18",

"ended": "2020-12-18",

"logo": "http://localhost:8000/media/E%3A/git/labs\_web/backend/media/Front\_eZs9usl.jpg",

"rating": 4

}

Для <http://localhost:8080/albums/>:

{

"title": "Some funny title",

"release\_date": "2020-12-18",

"cover": "http://localhost:8000/media/E%3A/git/labs\_web/backend/media/Front\_lNLPXFi.jpg",

"rating": 4,

"band": {

"id": 1

},

"genres": [

{

"id": 2

}

]

}

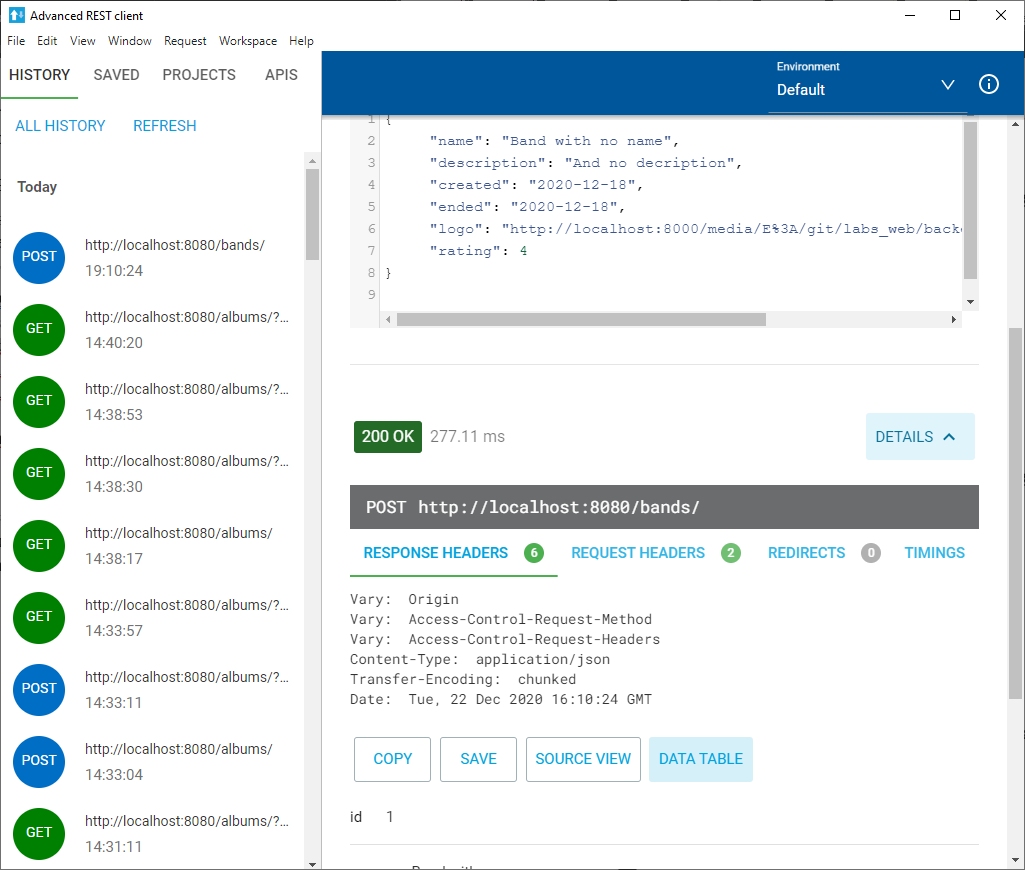
Для <http://localhost:8080/genres>:

{

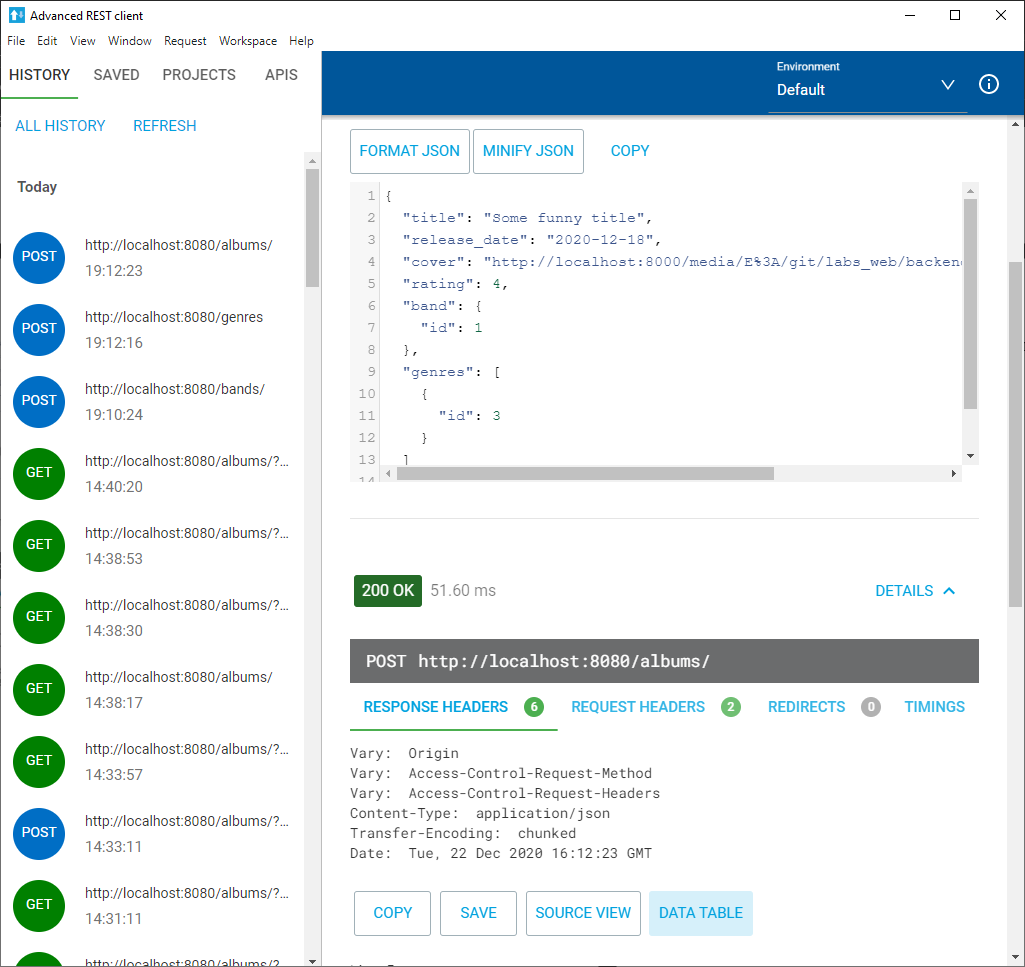
"name": "Жанр 1"

}

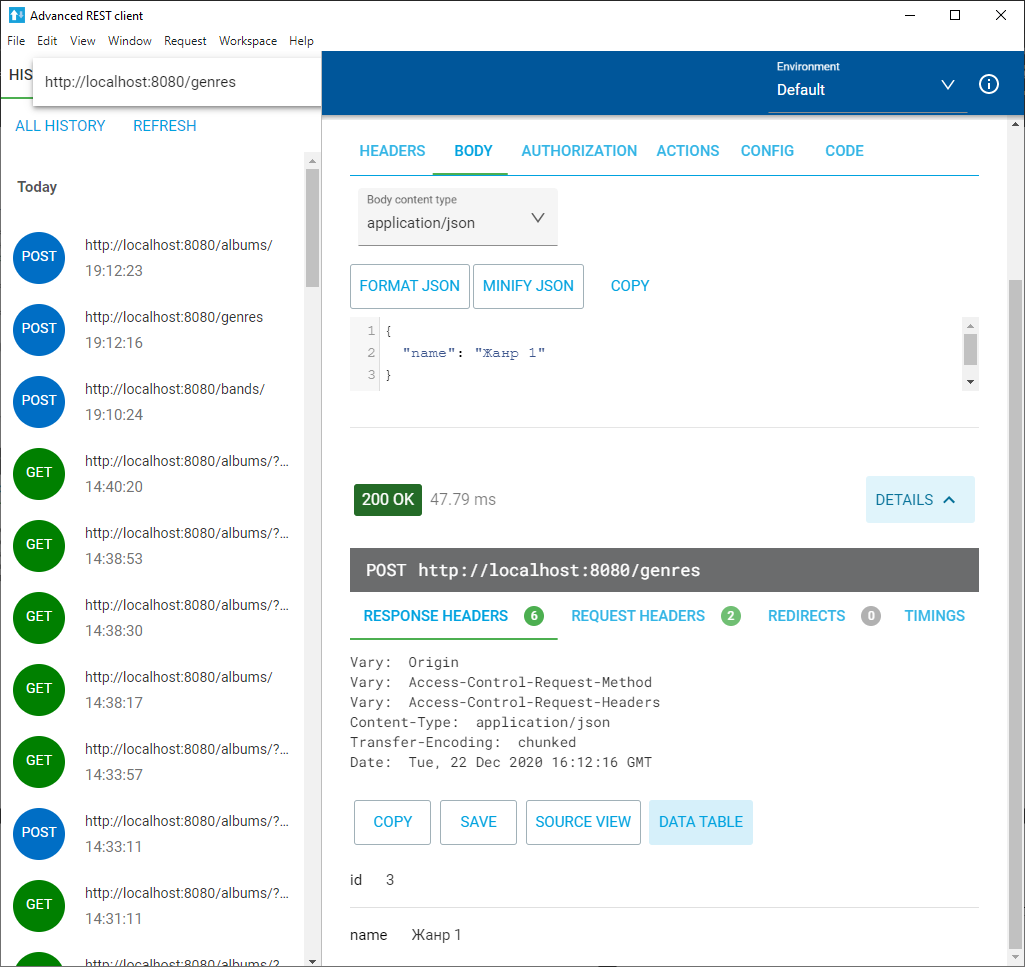
Выполним POST-запросы и проанализируем полученный результат.



*Рис. 4.1. Результат POST-запроса №1*



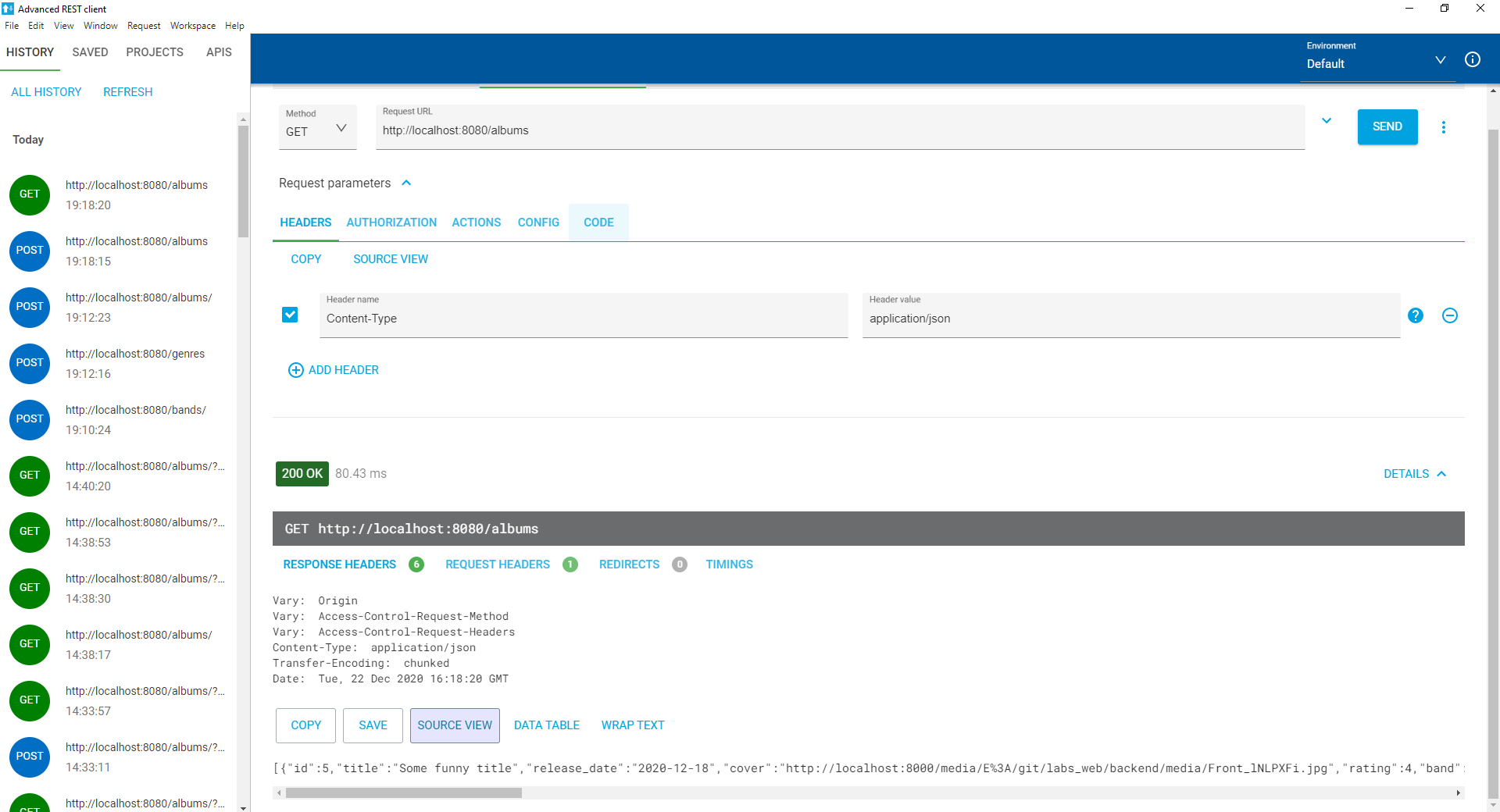
*Рис. 4.2. Результат POST-запроса №2*



*Рис. 4.3. Результат POST-запроса №3*

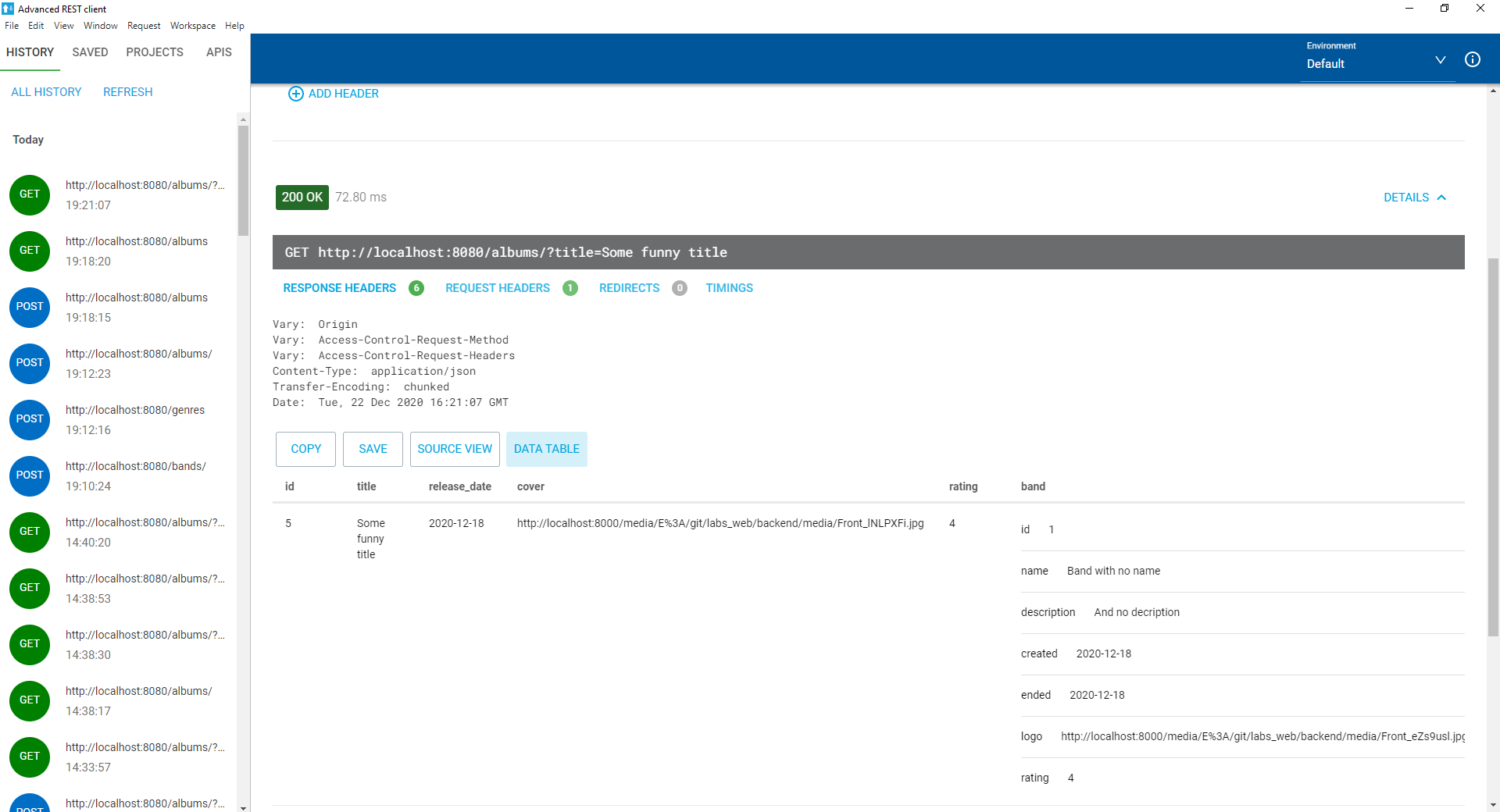
На каждый запрос от сервера был получен ответ 200 OK, что свидетельствует о том, что все запросы были выполнены успешно.

Поскольку теперь в базе данных уже содержатся некоторые данные, выполним GET-запрос по <http://localhost:8080/albums/>



*Рис. 4.4 Результат GET-запроса*

Был получен ответ 200 ОК, API вернула JSON с данными об альбомах. Добавим еще несколько альбомов и протестируем поиск по альбомам



*Рис. 4.5 Результат GET-запроса с параметром title*

Как и до этого, были получены корректные данные, из чего следует сделать вывод, что API функционирует верно.

В дальнейшем разработанное API будет использоваться frontend-приложением, реализованном на языке JavaScript с использованием библиотеки ReactJs.

# **Скриншоты**

# **Заключение**

# **Список литературы**

1. REST - Википедия. – [Электронный ресурс]. – URL: <https://ru.wikipedia.org/wiki/REST>
2. Введение в REST API — RESTful веб-сервисы. – [Электронный ресурс]. – URL: <https://habr.com/ru/post/483202/>
3. Spring Framework - Википедия – [Электронный ресурс]. – URL: https://ru.wikipedia.org/wiki/Spring\_Framework
4. Overview (Java platform SE 8). – [Электронный ресурс]. – URL: <https://docs.oracle.com/javase/8/docs/api/>
5. CRUD - Википедия. – [Электронный ресурс]. – URL: <https://ru.wikipedia.org/wiki/CRUD>
6. H2 Documentation. – [Электронный ресурс]. – URL: <https://www.h2database.com/javadoc/>
7. Spring Framework Documentation. – [Электронный ресурс]. – URL: <https://docs.spring.io/spring-framework/docs/current/reference/html/>
8. Spring Boot Reference Documentation. – [Электронный ресурс]. – URL: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>
9. API Help (Lombok). – [Электронный ресурс]. – URL: <https://projectlombok.org/api/help-doc.html>
10. Maven – Maven documentation. – [Электронный ресурс]. – URL: <https://maven.apache.org/guides/index.html>
11. Машнин Тимур Сергеевич. Технология Web-сервисов платформы Java. — БХВ-Петербург, 2012. — С. 115. — 560 с.

# **Приложения**