CZU: 004.65 DOI: 10.46727/c.steam-2023.p236-246

РАЗРАБОТКА Spring CRUD REST API ПРИЛОЖЕНИЙ С ВЫПОЛНЕНИЕМ ПРОВЕРКИ ОПЕРАЦИЙ CRUD ЧЕРЕЗ POSTMAN

Olga CERBU, dr. conf. universitar
https://orcid.org/0000-0002-6278-7115
Vitali CARAIVANOV
Государственный Университет Молдовы, Факультет Информатики

Резюме. В этой статье будет разработано простое приложение Spring Boot, которое предоставляет CRUD (Create, Read, Update, Delete) операции через REST API. Также будет показано, как проверить эти операции с использованием инструмента Postman.
Ключевые слова: Spring, Spring Boot, Spring Initializr, Spring Data JPA, CRUD, REST API, Postman, Postman Collections, GET, PUT, POST, UPDATE, DELETE, Controller, Entity, Repository.

Abstract. In this section, we will develop a simple Spring Boot application that provides CRUD (Create, Read, Update, Delete) operations through a REST API. We will also demonstrate how to verify these operations using the Postman tool.
Keywords: Spring, Spring Boot, Spring Initializr, Spring Data JPA, CRUD, REST API, Postman, Postman Collections, GET, PUT, POST, UPDATE, DELETE, Controller, Entity, Repository.

1. Создание проекта Spring Boot

Spring Boot — это проект в экосистеме Spring Framework, который предоставляет инструменты для создания легких, самодостаточных и быстрых приложений на языке Java. Spring Boot упрощает конфигурацию, включает встроенные серверы приложений, предоставляет автоконфигурацию, упрощает управление зависимостями, обеспечивает модульность, включает встроенные инструменты разработки и поддерживает микросервисную архитектуру. Этот фреймворк популярен среди разработчиков и облегчает создание качественных приложений с минимальными усилиями.

Необходимо создать новый проект Spring Boot при помощи Spring Initializr. Для этого необходимо перейти на ссылку https://start.spring.io и сконфигурировать проект согласно нуждам.

Здесь можно выбрать версию Java, Spring Boot, указать различные метаданные, такие, как имя проекта, его описание, имя пакета, в котором будет располагаться проект, и т.д., систему сборки проекта.

Для данной статьи была выбрана Java 17-ой версии, а Spring Boot версии 3.1.5. Следующим шагом будет выбор зависимостей для проекта, которые необходимы для полноценной функциональности приложения.

Рисунок 1. Окно конфигурации проекта Spring Initialzr

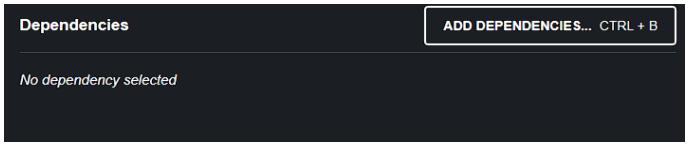

Рисунок 2. Окно выбора зависимости проекта через Spring Initialzr

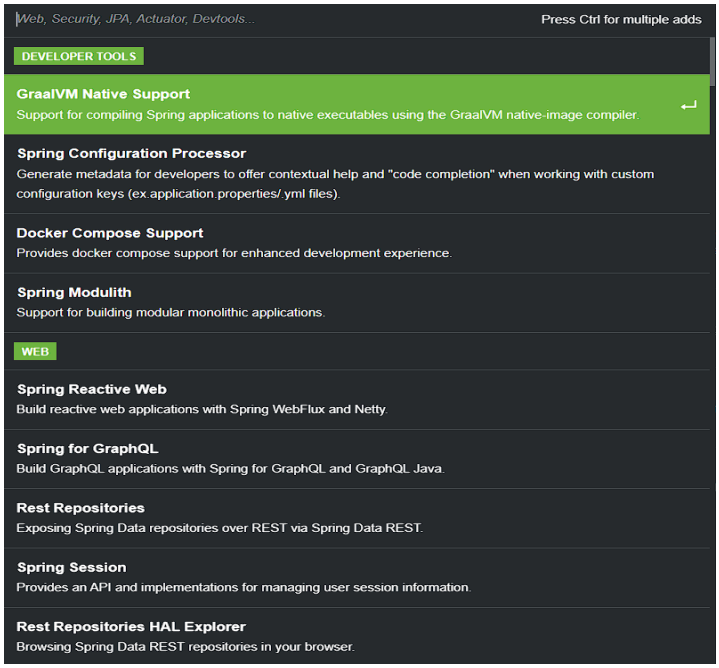

Рисунок 3. Доступные зависимости для проекта, создаваемого при помощи Spring Initialzr

Изначально, список выбранных зависимостей пуст. Для добавления новой зависимости необходимо нажать на кнопку «ADD DEPENDENCIES», либо нажать комбинацию клавиш CTRL + B.

После нажатия на кнопку для добавления зависимостей будет представлена когорта зависимостей, как на **рисунке 3**.

Здесь также представлен поиск зависимостей по ключевым словам.

Конкретно в этом примере задействованы следующие зависимости:

- **Spring Data JPA** – сохранение данных в SQL-хранилищах с использованием Java Persistence API с помощью Spring Data и Hibernate.
- **H2 Database** – предоставляет быструю встроенную базу данных, поддерживающую JDBC API и доступ через R2DBC, с небольшим размером (2 МБ). Поддерживает встроенный и серверный режимы, а также приложение с консолью на основе браузера.
- **Spring WEB** -создание веб-приложений, включая RESTful, с использованием Spring MVC. В качестве встроенного контейнера по умолчанию используется Apache Tomcat.
- **Spring Boot DevTools** – обеспечивает быструю перезагрузку приложения, LiveReload и настройки для улучшенного опыта разработки.

- **Lombok** – библиотека аннотаций для языка Java, которая помогает уменьшить избыточный код.

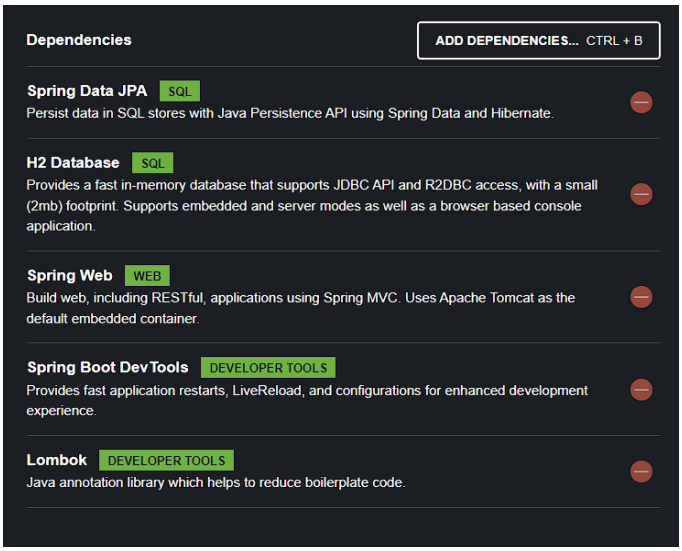

Рисунок 4. Выбранные зависимости для примера Spring CRUD REST API приложения

Список выбранных зависимостей представлен на **рисунке 4**.

После выбора зависимостей, необходимо нажать на кнопку «GENERATE», и сконфигурированный проект будет загружен на локальный компьютер. Его необходимо открыть через среду разработки для дальнейшей работы.

2. Создание простого приложения с использованием CRUD операций

Открыв сгенерированный проект через среду разработки IntelliJ IDEA, необходимо перейти в файл свойств *application.properties*, где необходимо указать конфигурацию для базы данных H2 (которая задействована в рамках примера, в реальных же проектах, это может быть любая другая база данных). Конфигурация из примера представлена в **листинге 1**.

```
# Листинг 1. application.properties, конфигурация
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

```
package com.example.spring_crud.repository;

import com.example.spring_crud.entity.Task;
import org.springframework.data.jpa.repository.JpaRepository;
// Листинг 2. Объявление интерфейса TaskRepository
public interface TaskRepository extends JpaRepository<Task, Long> {
}
```

Далее будут следовать листинги с кодом приложения, и его разъяснением.

В интерфейсе *TaskRepository* определены методы для выполнения стандартных операций базы данных (например, создание, чтение, обновление и удаление записей) для сущности Task. Он расширяет интерфейс JpaRepository, предоставляемый Spring Data JPA, и использует два параметра типа:

Task: это класс сущности, для которой создается репозиторий. Репозиторий будет использоваться для доступа к данным этой сущности в базе данных.

Long: это тип данных для идентификатора сущности. В данном случае, ожидается, что идентификатор сущности Task будет числовым (Long).

Этот интерфейс не содержит явной реализации методов, так как Spring Data JPA автоматически генерирует реализацию на основе соглашений и именованных запросов, используя концепцию репозитория. Это позволяет легко выполнять операции CRUD (Create, Read, Update, Delete) сущности Task в базе данных без необходимости написания SQL-запросов вручную.

```
// Листинг 3. Объявление класса – сущности Task
package com.example.spring_crud.entity;

import 4ombok4.persistence.Entity;
import 4ombok4.persistence.GeneratedValue;
import 4ombok4.persistence.GenerationType;
import 4ombok4.persistence.Id;
import 4ombok.Data;
import 4ombok.NoArgsConstructor;

@Entity
@Data
@NoArgsConstructor
public class Task {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String description;


}
```

Этот код описывает класс сущности Task, который будет использоваться в Spring приложении.

@Entity -эта аннотация обозначает, что класс Task является сущностью, которая будет отображена в базе данных.

@Data – это аннотация Lombok, которая автоматически создает геттеры, сеттеры, методы equals(), hashCode(), и toString(), что уменьшает объем шаблонного кода.
@NoArgsConstructor – эта аннотация Lombok создает конструктор без аргументов для класса Task, что полезно при работе с JPA и Spring.
@Id – эта аннотация обозначает поле id как идентификатор сущности.
@GeneratedValue(strategy = GenerationType.IDENTITY) – эта аннотация указывает, что значение поля id будет автоматически генерироваться с использованием стратегии GenerationType.IDENTITY, что обычно соответствует автоинкрементному полю в базе данных.
private Long id – это поле для хранения уникального идентификатора задачи.
private String title – это поле для хранения заголовка задачи.
private String description – это поле для хранения описания задачи.
Этот класс представляет собой JavaBean, который соответствует таблице в базе данных и используется для хранения и работе с данными задачи в Spring приложении.

```
// Листинг 4 Класс контроллер TaskController
package com.example.spring_crud.controller;

import com.example.spring_crud.entity.Task;
import com.example.spring_crud.repository.TaskRepository;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/tasks")
public class TaskController {
    private final TaskRepository taskRepository;

    public TaskController(TaskRepository taskRepository) {
        this.taskRepository = taskRepository;
    }

    @PostMapping
    public Task createTask(@RequestBody Task task) {
        return taskRepository.save(task);
    }

    @GetMapping
    public List<Task> getAllTasks() {
```

```
        return taskRepository.findAll();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Task> getTaskById(@PathVariable Long id) {
        Task task = taskRepository.findById(id).orElse(null);
        if (task != null) {
            return ResponseEntity.ok(task);
        } else {
            return ResponseEntity.notFound().build();
        }
    }
    @PutMapping("/{id}")
    public ResponseEntity<Task> updateTask(@PathVariable Long id, @RequestBody Task updatedTask)
{
        Task task = taskRepository.findById(id).orElse(null);
        if (task != null) {
            task.setTitle(updatedTask.getTitle());
            task.setDescription(updatedTask.getDescription());
            taskRepository.save(task);
            return ResponseEntity.ok(task);
        } else {
            return ResponseEntity.notFound().build();
        }
    }
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteTask(@PathVariable Long id) {
        taskRepository.deleteById(id);
        return ResponseEntity.noContent().build();
    }
}
```

Этот код представляет собой REST-контроллер для операций CRUD (Create, Read, Update, Delete) с сущностью Task.

@RestController – эта аннотация обозначает класс как контроллер, который обрабатывает HTTP-запросы.

@RequestMapping(«/api/tasks») – эта аннотация указывает, что все методы контроллера будут обрабатывать запросы, начинающиеся с /api/tasks.

@Autowired - используется для внедрения зависимости TaskRepository в контроллер через конструктор.

@PostMapping – этот метод обрабатывает HTTP POST-запросы и создает новую задачу, используя данные, полученные из тела запроса.

@GetMapping – этот метод обрабатывает HTTP GET-запросы и возвращает список всех задач.
@GetMapping(«/{id}») – этот метод обрабатывает HTTP GET-запросы с параметром id в URL и возвращает задачу по указанному идентификатору.
@PutMapping(«/{id}») – этот метод обрабатывает HTTP PUT-запросы и обновляет существующую задачу по указанному идентификатору.
@DeleteMapping(«/{id}») – этот метод обрабатывает HTTP DELETE-запросы и удаляет задачу по указанному идентификатору.
Этот контроллер предоставляет простой API для выполнения операций CRUD с задачами и использует TaskRepository для взаимодействия с базой данных.

```
// Листинг 5 Точка запуска Spring Boot приложения
package com.example.spring_crud;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringCrudApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringCrudApplication.class, args);
    }

}
```

Этот код представляет собой класс, который является входной точкой Spring Boot приложения.
@SpringBootApplication – эта аннотация указывает, что класс SpringCrudApplication является точкой входа в Spring Boot приложение. Она включает автоконфигурацию Spring Boot, сканирование компонентов и настройку приложения.
Когда приложение запущено, оно начинает обработку HTTP-запросов и выполняет бизнес-логику вашего приложения.
Примечание: Ваши наименования классов, а также пути, указанные в package, могут отличаться от тех, что указаны в текущем примере. В рамках простого CRUD приложения использованы текущие имена пакетов и классов.
После написания кода, приложение необходимо запустить, и проверить работу методов контроллера.

3. Проверка методов контроллера через Postman

Postman — это популярная утилита для тестирования и разработки веб-сервисов и RESTful API. Она предоставляет мощные инструменты для отправки HTTP-запросов, проверки ответов и автоматизации процессов тестирования API. Postman позволяет создавать и отправлять различные типы HTTP-запросов, такие как GET, POST, PUT, DELETE, и многие другие.

Утилита предоставляет удобный редактор запросов с подсветкой синтаксиса и автозаполнением, помогая создавать запросы и настраивать заголовки, параметры и тело запроса. Postman позволяет создавать среды для управления переменными и настройками, что упрощает работу с разными средами (например, тестовая среда, продакшн и другие), а также предоставляет возможность для группировки запросов в коллекции для организации тестовых сценариев и рабочих процессов, и т.д.

Примечание: Процесс регистрации на официальном сайте Postman будет упущен.

В самом инструменте была создана коллекция *Spring CRUD REST API Test*, которая включает себя четыре запроса, соответствующие операциям CRUD – Create (POST), Read (GET), Update (PUT), Delete (DELETE) с соответствующими названиями для большего удобства, а также создана переменная в рамках коллекции BASE_URL, которая включает себя общий для всех запросов URL адрес.

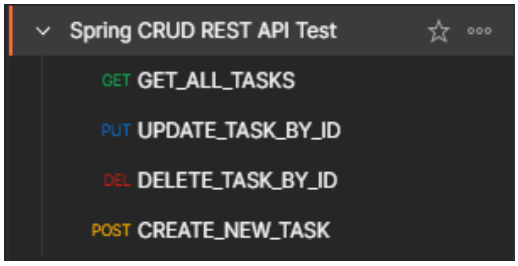

Рисунок. 5. Коллекция запросов Spring CRUD REST API TEST

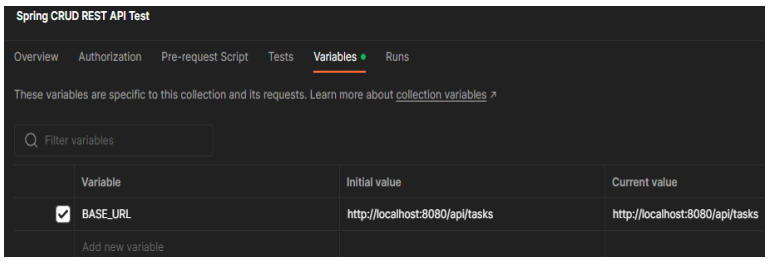

Рисунок. 6. Список переменных в коллекции *Spring CRUD REST API Test*

Для выполнения запросов необходимо обращаться к конкретным *endpoint*[1] с различными параметрами запроса. Все запросы будут выполнены для проверки функциональности методов контроллера *TaskController*.

В качестве тела в POST запрос посылается в формате *JSON*[2] объект Task без поля id (так как оно генерируется и инкрементируется автоматически). После выполнения POST запроса вернется созданный объект Task в теле ответа. Запрос GET вернет в теле ответа список всех Task.

[1] endpoint - конечная точка или URL (Uniform Resource Locator), по которой можно получить доступ к какому-либо сервису или ресурсу

[2] JSON (JavaScript Object Notation) — это легковесный формат обмена данными, который часто используется в веб-разработке.

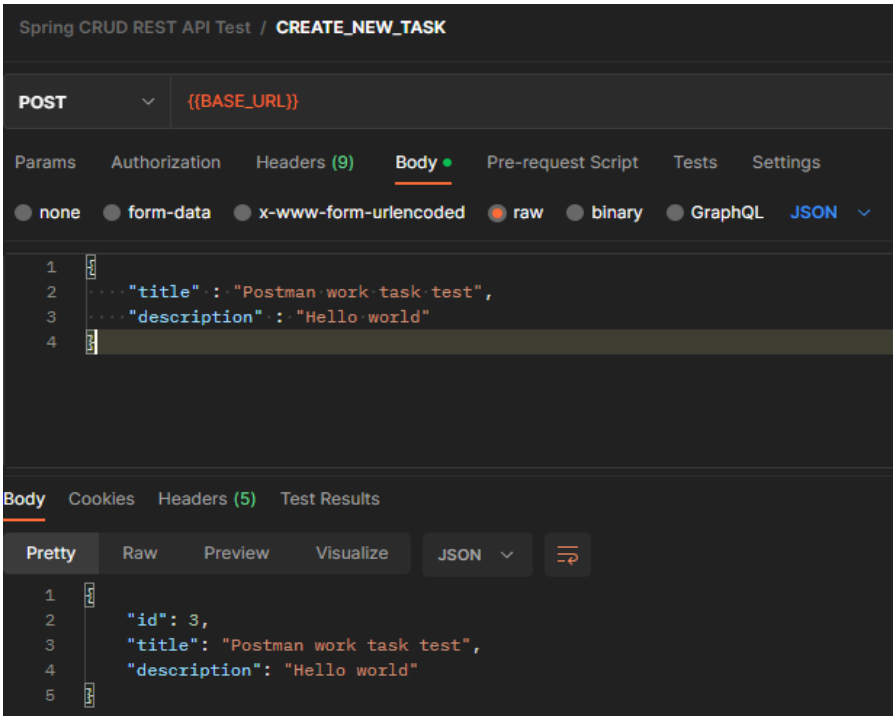

Рисунок 7. Структура POST запроса и результат его выполнения

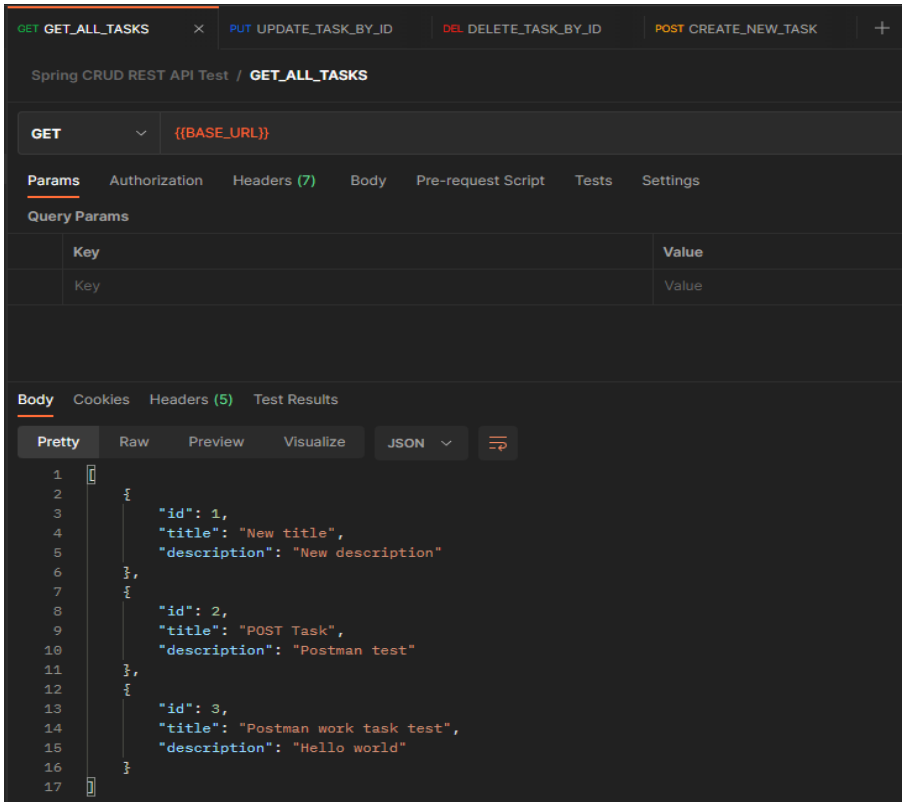

Рисунок 8. Выполнение GET запроса на получение всех Task

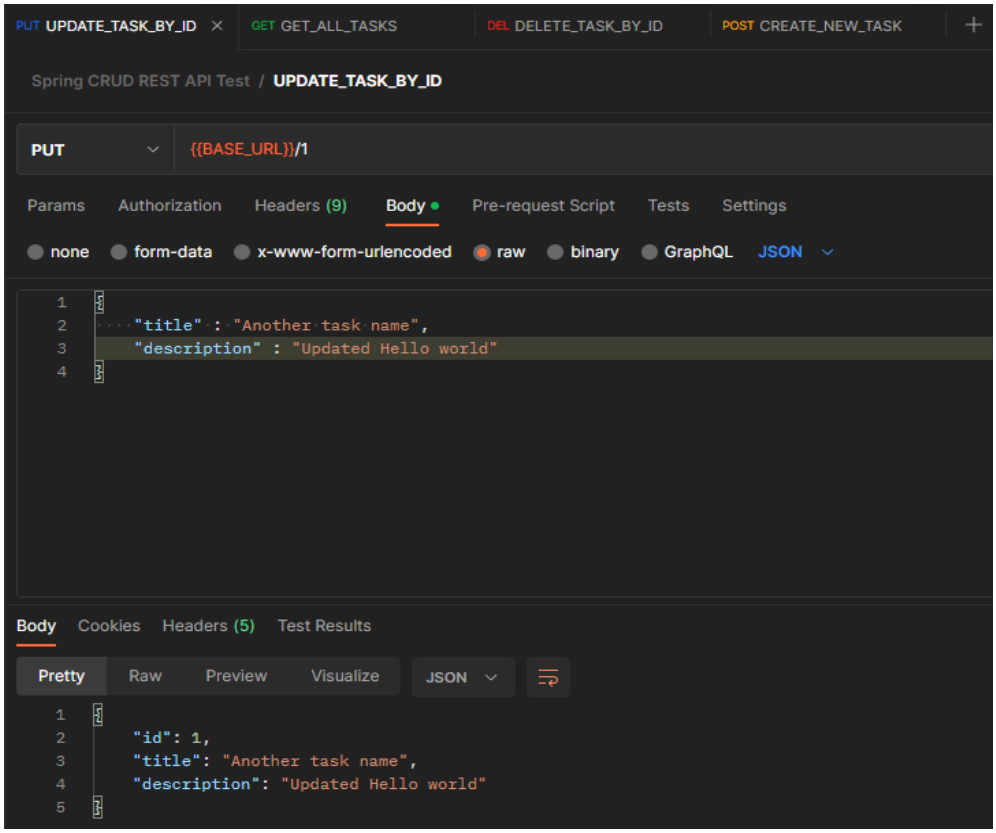

Рисунок 9. Выполнение PUT запроса на обновление Task по id = 1

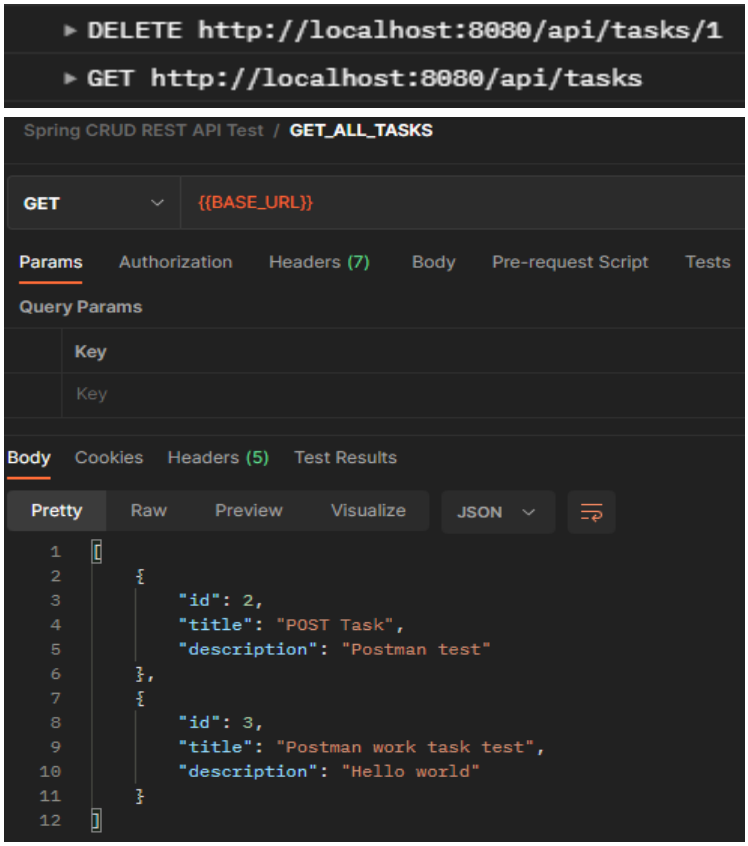

Рисунок 10. Выполнение DELETE запроса на удаление Task с id = 1. Выполнение GET запроса на отображение всех объектов Task после выполнение DELETE

Выводы

В данной работе указанно как Вы можете подключить Postman к своим рабочим процессам API с помощью интеграции с популярными сторонними решениями и как использовать интеграцию для автоматического обмена данными между Postman и другими инструментами, которые вы используете для разработки API.

Библиография

1. WALLS, C. *Spring in Action*, 6th Edition.
2. Spring Framework Documentation https://docs.spring.io/spring-framework/reference/index.html
3. Postman Documentation https://learning.postman.com/docs/introduction/overview/