

**Task 1: Java REST API Example.**

Implement an application in java which provides a REST API with endpoints for searching, creating and deleting “server” objects:

* GET servers: Should return all the servers if no parameters are passed. When server id is passed as a parameter – return a single server or 404 if there’s no such a server.
* PUT a server: The server object is passed as a json-encoded message body. Here’s an example:

{

“name” : “my centos”,

“id” : “123”,

“language” : “java”,

“framework” : “Django”

}

* DELETE a server: The parameter is a server ID.
* GET (find) servers by name: The parameter is a string. Must check if a server name contains this string and return one or more servers found. Return 404 if nothing is found.

“Server” objects should be stored in MongoDB database.

Be sure that you can show how your application responds to requests using postman, curl or any other HTTP client.

**Solution 1:**

**SOFTWARE USED:** Spring Tools Suite 3.9.12, MongoDB Community Server, Mongo Shell, MongoDB Compass.

**DATABASE USED:** MongoDB

**FRAMEWORK USED:** Spring Framework

**STEPS:**

1. At first, create a **Spring Boot** project.

Spring Boot makes it easy to create stand-alone, production-grade spring based applications that you can just-run.

It has embedded Tomcat, Jetty or Undertow directly.

It provides opinionated ‘starter’ dependencies to simplify your build configuration.

Automatically configures Spring and 3rd party libraries whenever possible.

It also provides production-ready features such as metrics, health checks, and externalized configuration.

1. Add “**Spring Web, Spring Boot DevTools, Spring Data MongoDB, Validation**” dependencies to the project.
2. I have followed **Spring MVC Architecture** to implement the above task.

Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.

MVC Mean:

Model – Contains the data of the application. A data can be a single object or a collection of objects.

Controller – Contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.

@View – A view represents the provided information in a particular format.

Note: View pages are not added but the data can be viewed as JSON Format through REST APIs.

1. After the successful creation of the project, your configuration file (pom.xml) should look like this:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<project xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.7.5</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>com.krishnaveni.ass</groupId>

<artifactId>SK-1</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>SK-1</name>

<description>Spring Boot </description>

<properties>

<java.version>11</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-mongodb</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<scope>provided</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

1. Now, create the **model, controller, exception, and repository packages.**
2. Firstly, create the model class with some attributes and mark it with **private access modifier.** Add **validation annotations** to the attributes for data consistency.
3. Next, create a repository and extend the parent class **MongoRepository** as our data should be stored in MongoDB database. It takes two arguments. One is your model class and the second is your datatype of the primary key.
4. Next, create an exception class and extend the parent class **Exception/Runtime Exception.** Create a parameterized constructor to display the respective exceptional message.
5. Now, create a controller class and mark it with **@RestController annotation** and create the methods for all CRUD operations and annotate them with the HTTP Requests Mappings and add the respective urls.
6. At last, add your MongoDB credentials and database name and other info in **application.properties file.**

spring.data.mongodb.host=Hostname

spring.data.mongodb.port=PortNumber

spring.data.mongodb.database=DatabaseName

1. Now, run the project. Select the project, and right click on it and choose to run as **Spring Boot App.** Before running the project, make sure that the database server is running already.
2. To run the database server, open the **mongod.exe** from the **MongoDB Community Server** bin folder. Then, open the **mongosh.exe** from the **Mongo shell** bin folder. Copy the **Connection string** in the **Mongo shell window.**
3. Open the MongoDB Compass. Create a new connection. Paste the copied string there and choose **Connect.** Now, your database connection is successfully established.

**TESTING THROUGH POSTMAN:**

1. **Save the data**

**INPUT:**

**Method: POST**

**Url:** [**http://localhost:8080/addApp**](http://localhost:8080/addApp)

**Body:**

{

        "id": 1,

        "appName": "Facebook",

        "appDeveloper": "Mark Zuckerberg and Eduardo Saverin",

        "introYear": 2010,

        "pl": "C++, Hack",

        "appType": "Online Social Media app"

    }

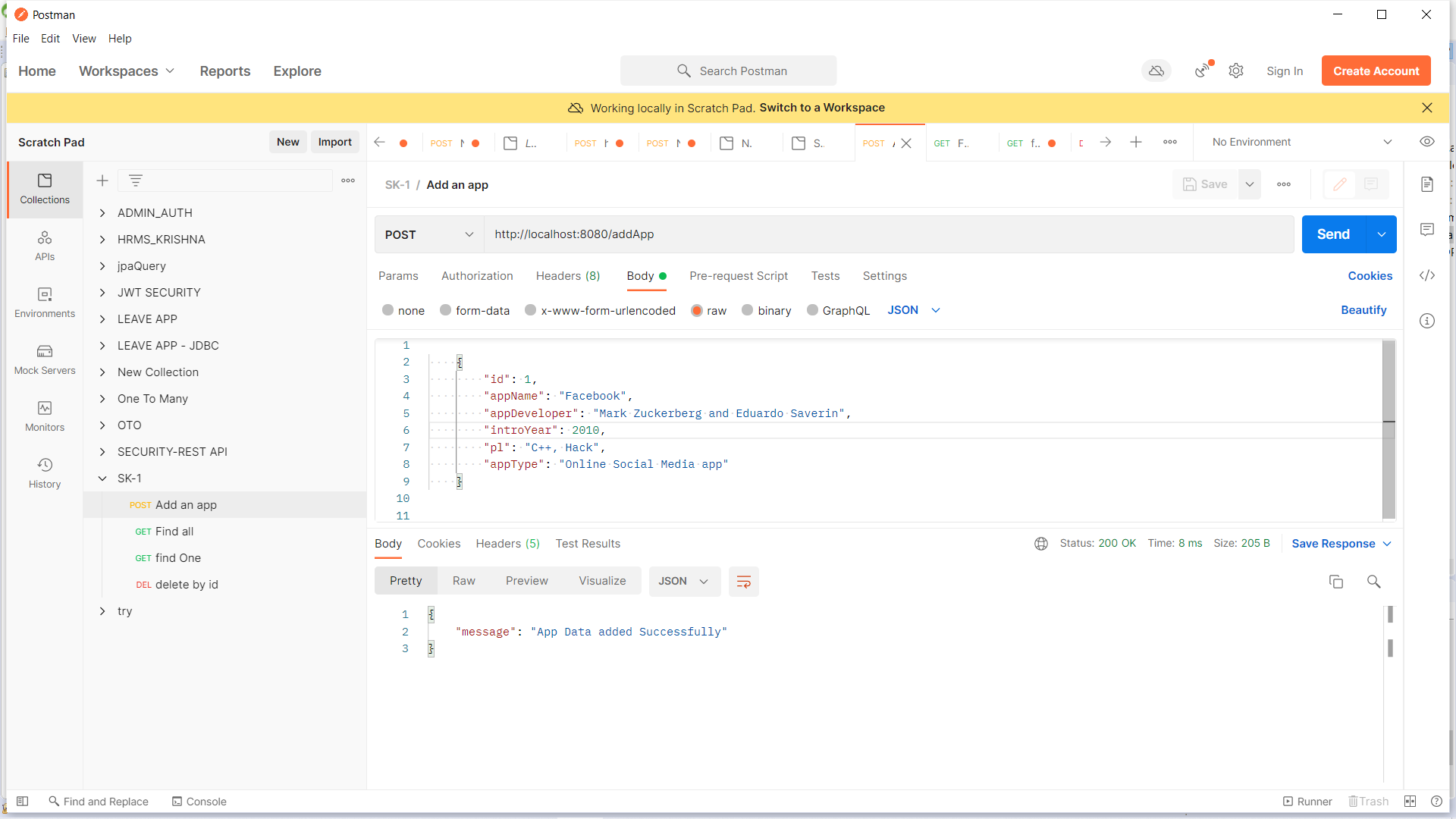
Accepts the data as JSON Format through Body section of the editor.

Then click on Send.

**OUTPUT:**

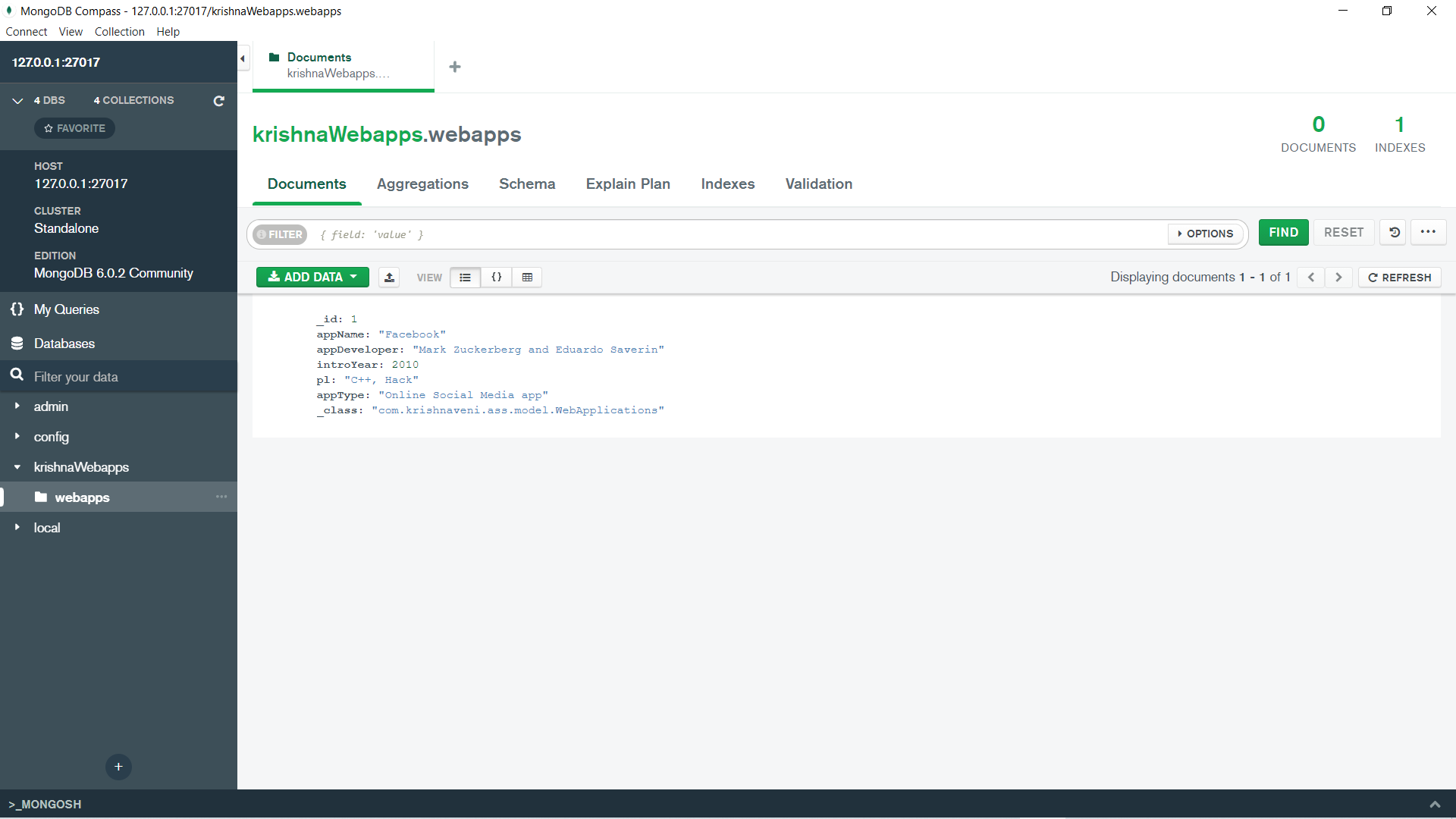
A message is displayed based on the response of the request.

The data has been added successfully. So, the message **“App data added successfully”** is displayed.

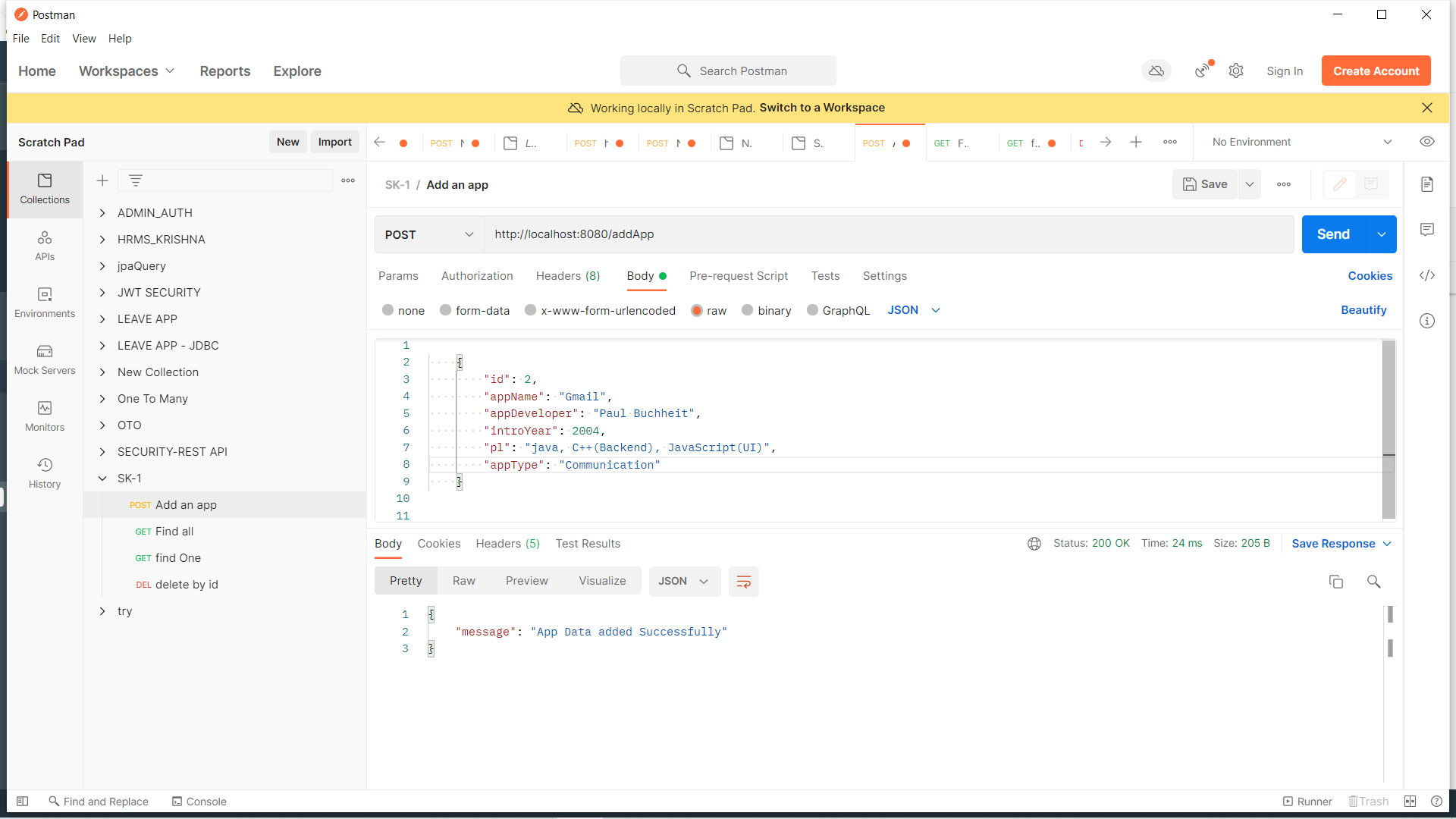


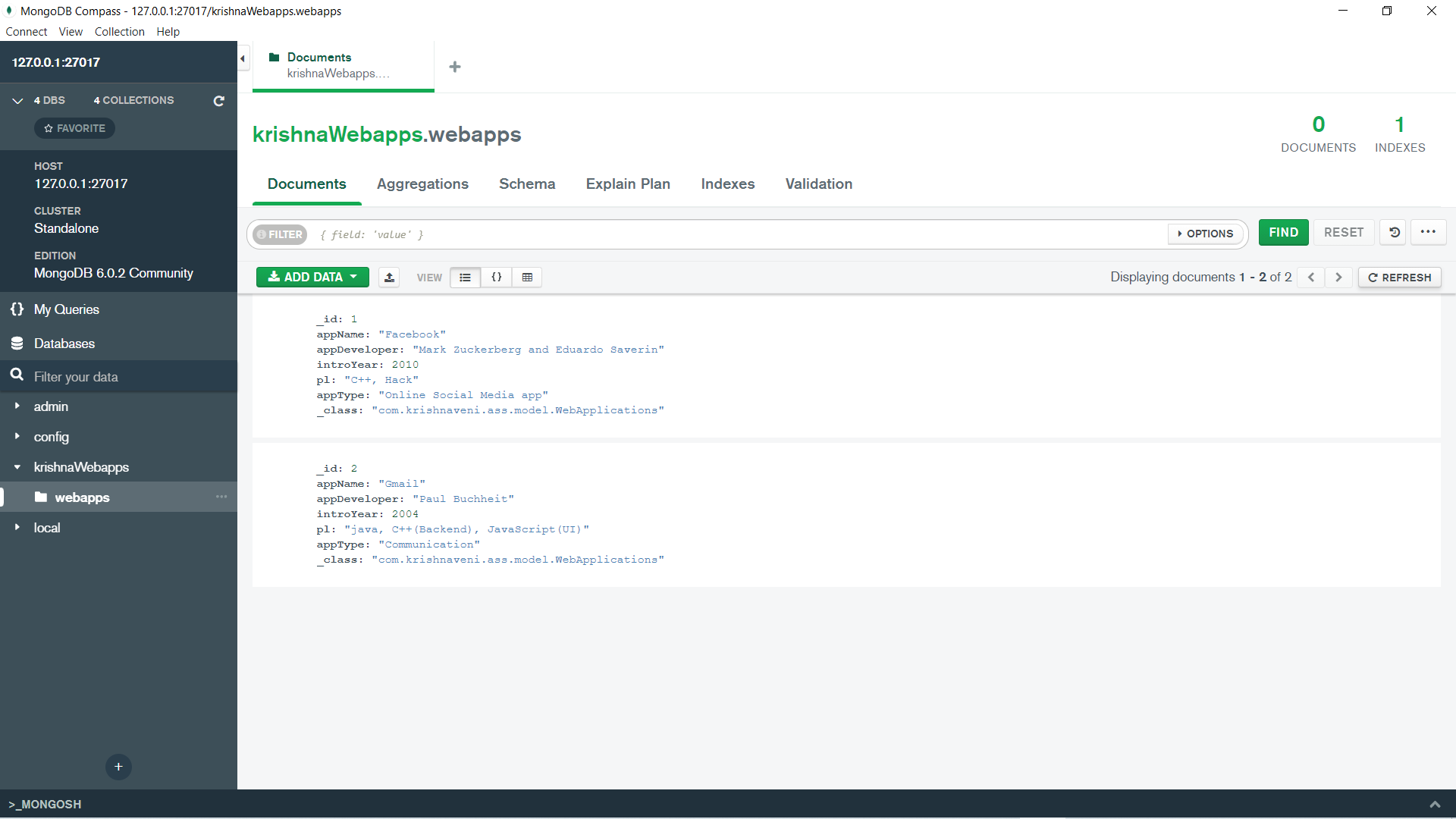
Check the database:

Open the **webapps** collection and go to **Documents** section to view the data.



Adding some more data to perform some more operations.



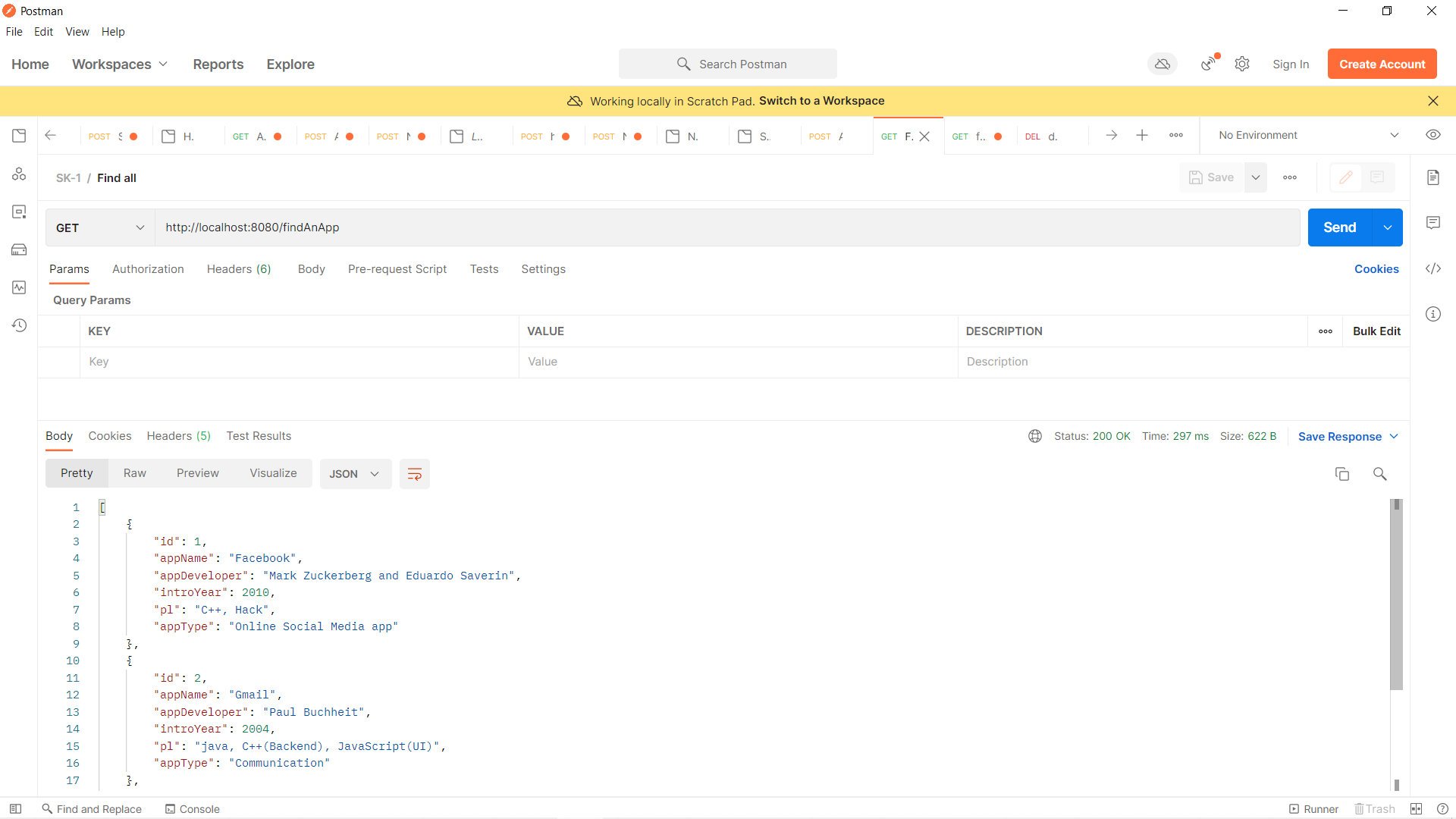


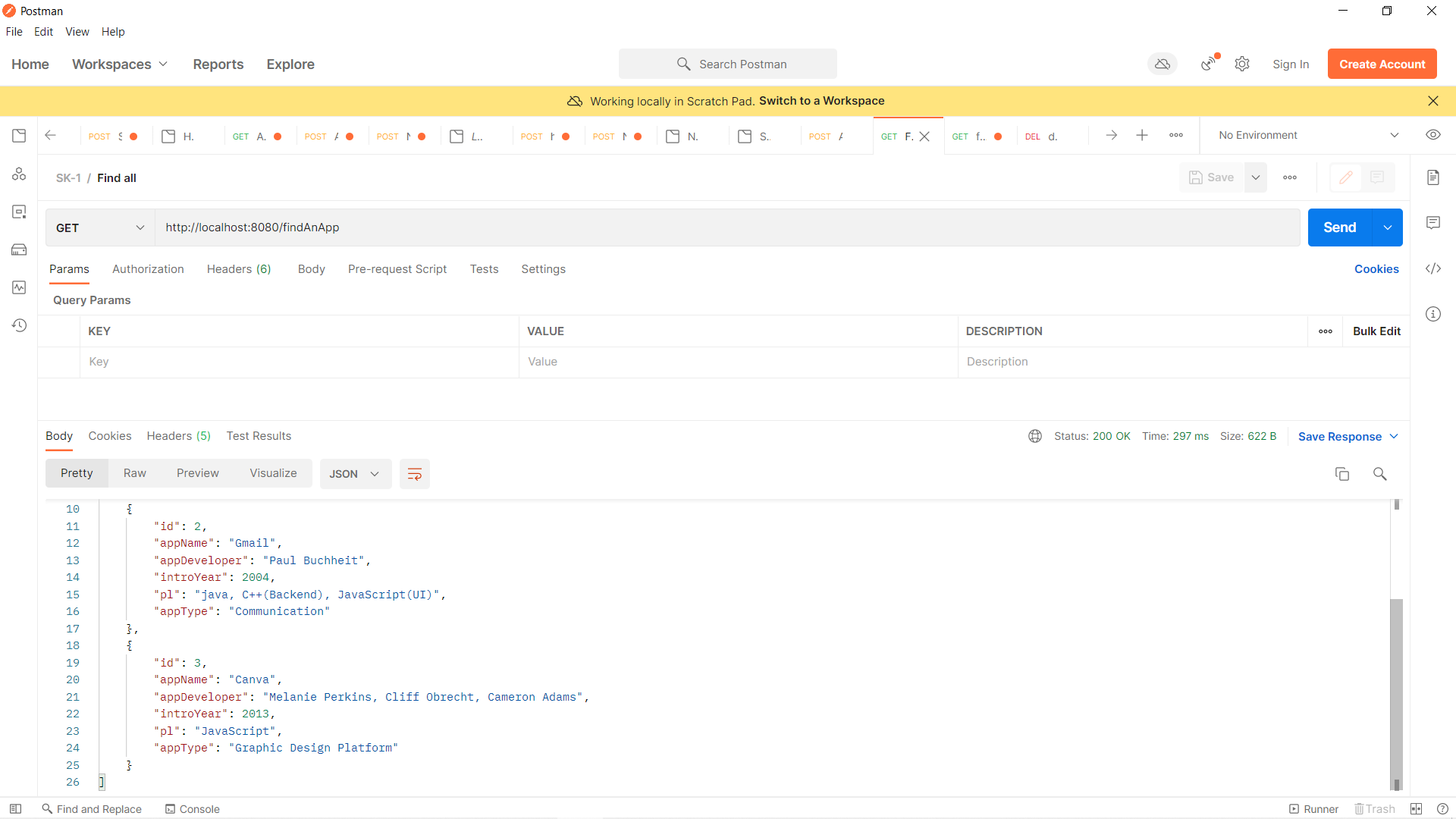
1. **Find all the data**

**INPUT:**

**Method: GET, Url:** [**http://localhost:8080/findAnApp**](http://localhost:8080/findAnApp)

**OUTPUT:**





1. **Find the data if parameter “id” is passed:**

**INPUT:**

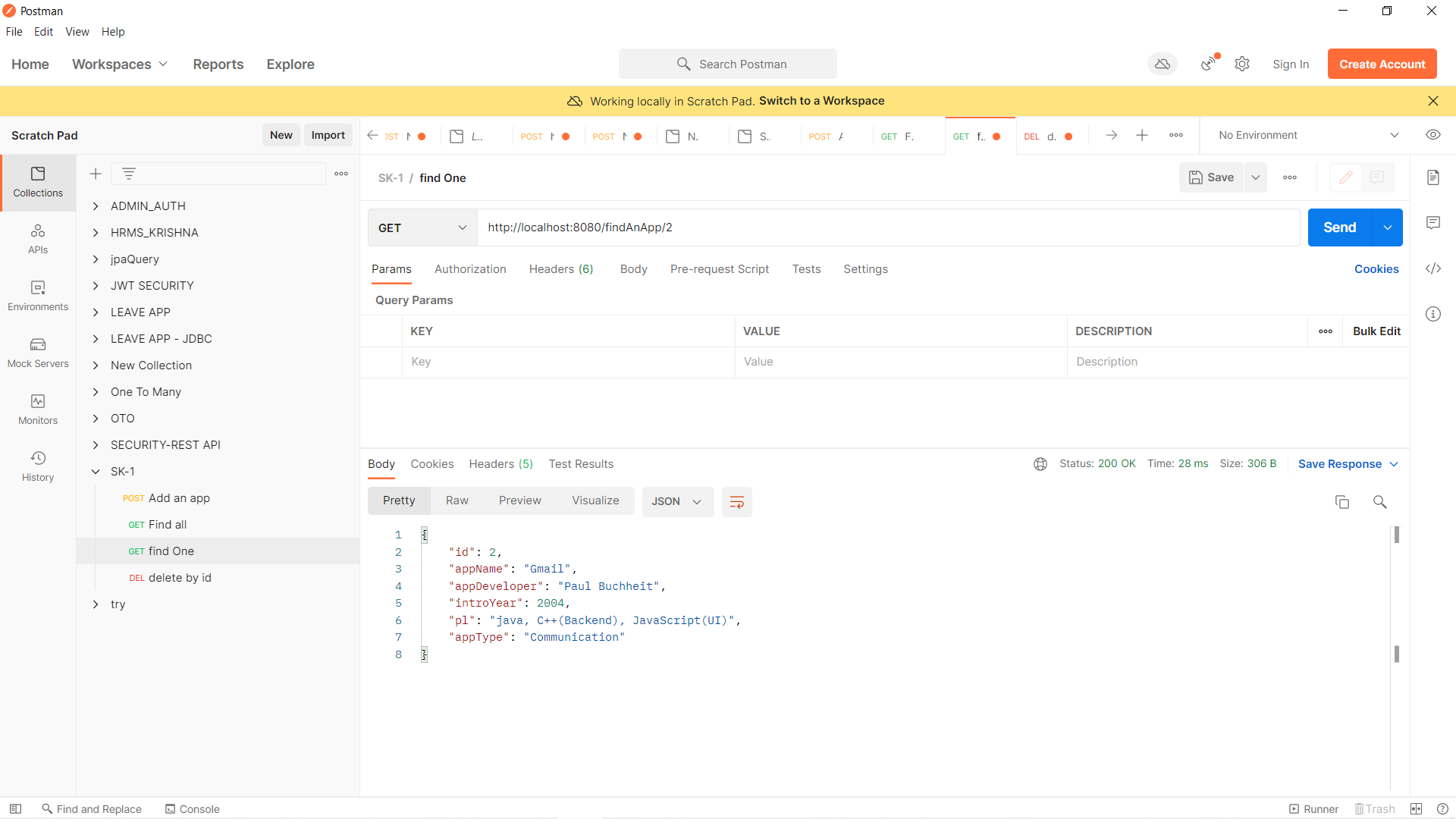
**Method: GET**

**Url:** [**http://localhost:8080/findAnApp/2**](http://localhost:8080/findAnApp/2)

**OUTPUT:**

**Http Status: 200 OK**

The data stored with **id:2** details are displayed in the **Body section** of the **Response.**



1. **Find by “id”, if no app found:**

**INPUT:**

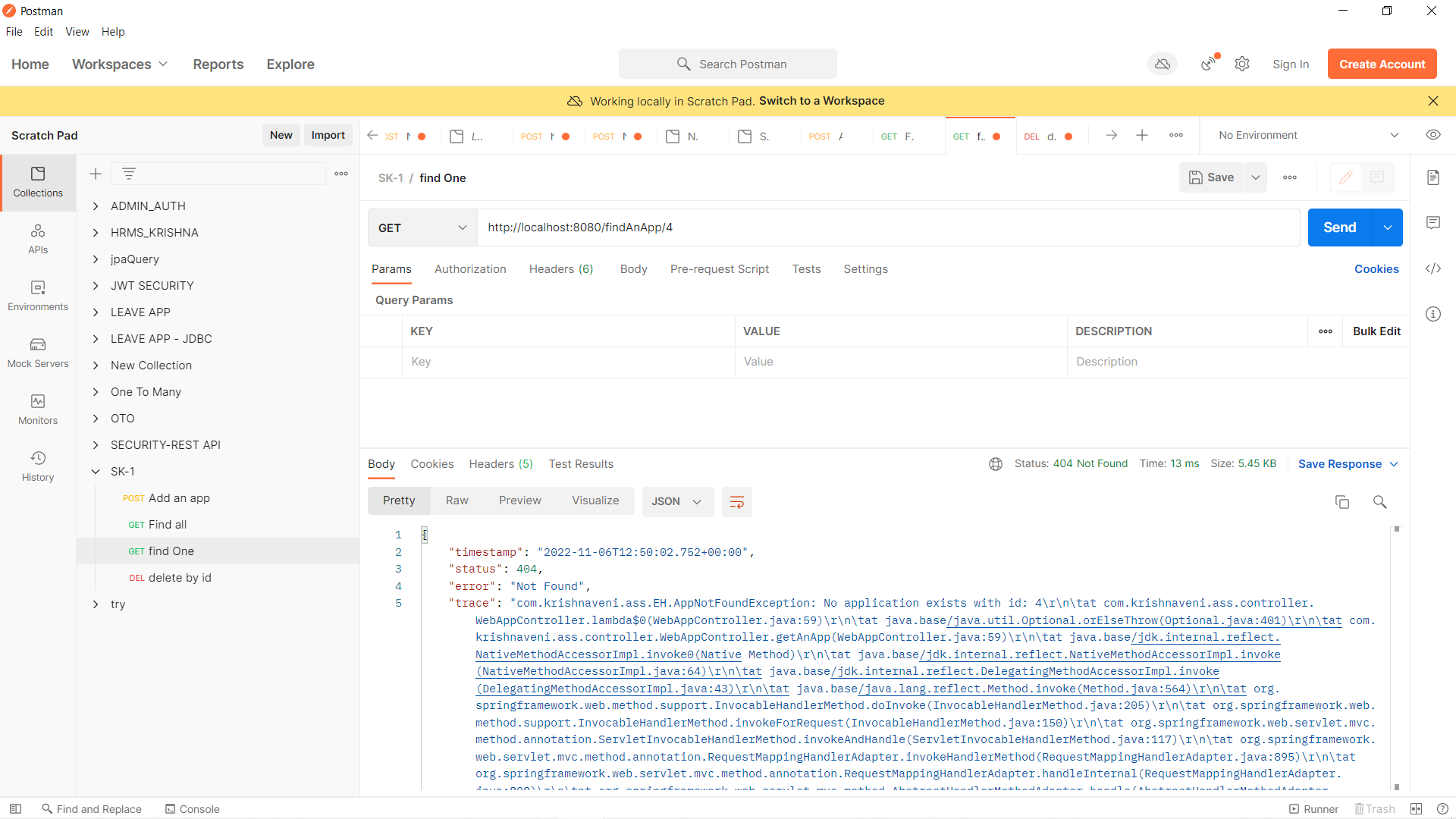
**Method: GET**

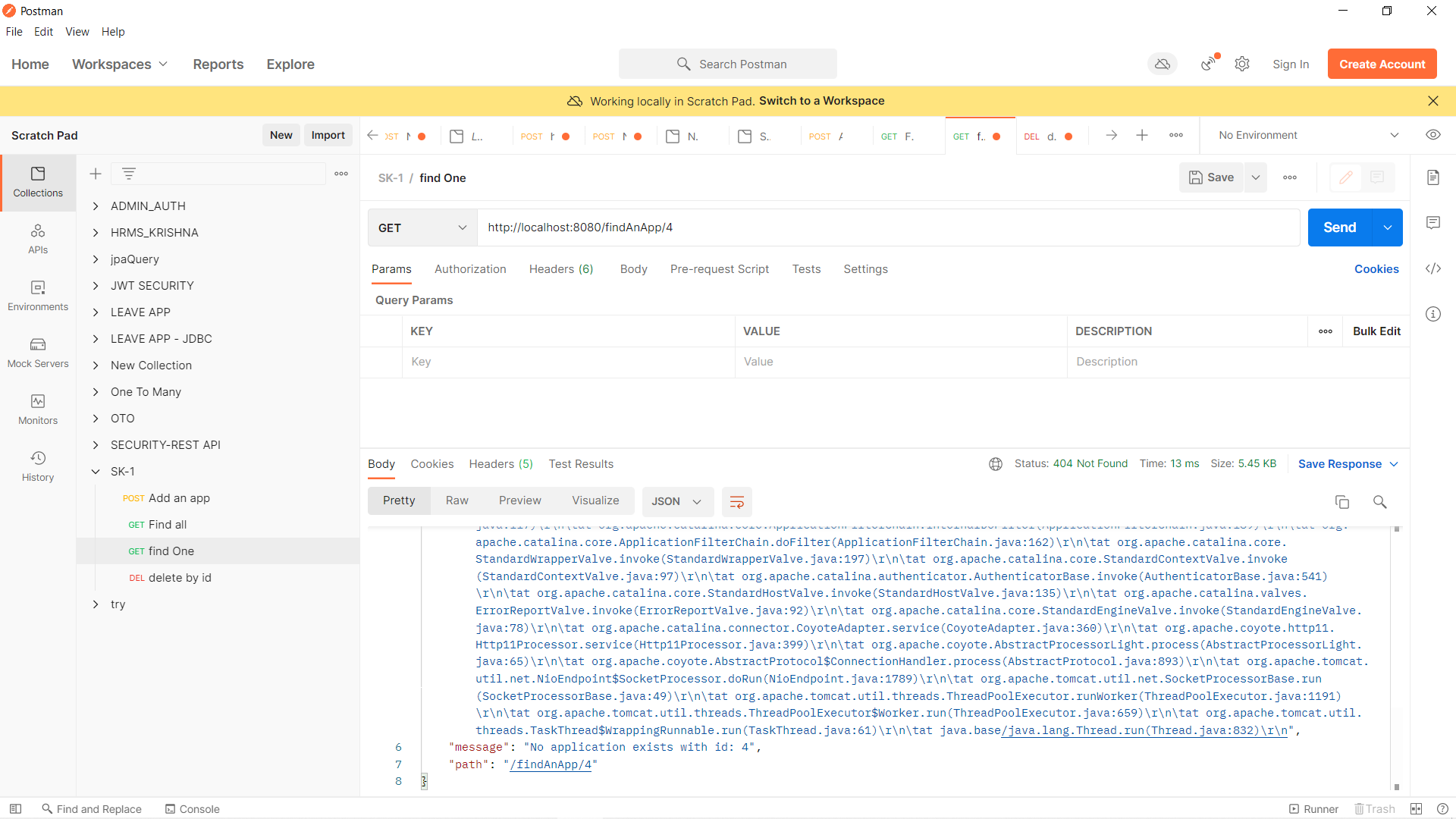
**Url:** [**http://localhost:8080/findAnApp/4**](http://localhost:8080/findAnApp/42)

**OUTPUT:**

**Http Status: 404 Not Found**

The data with **id:4** details is not found in the database. So, AppNotFoundException details are displayed in the **Body section** of the **Response.**





1. **Find by name:**

**INPUT:**

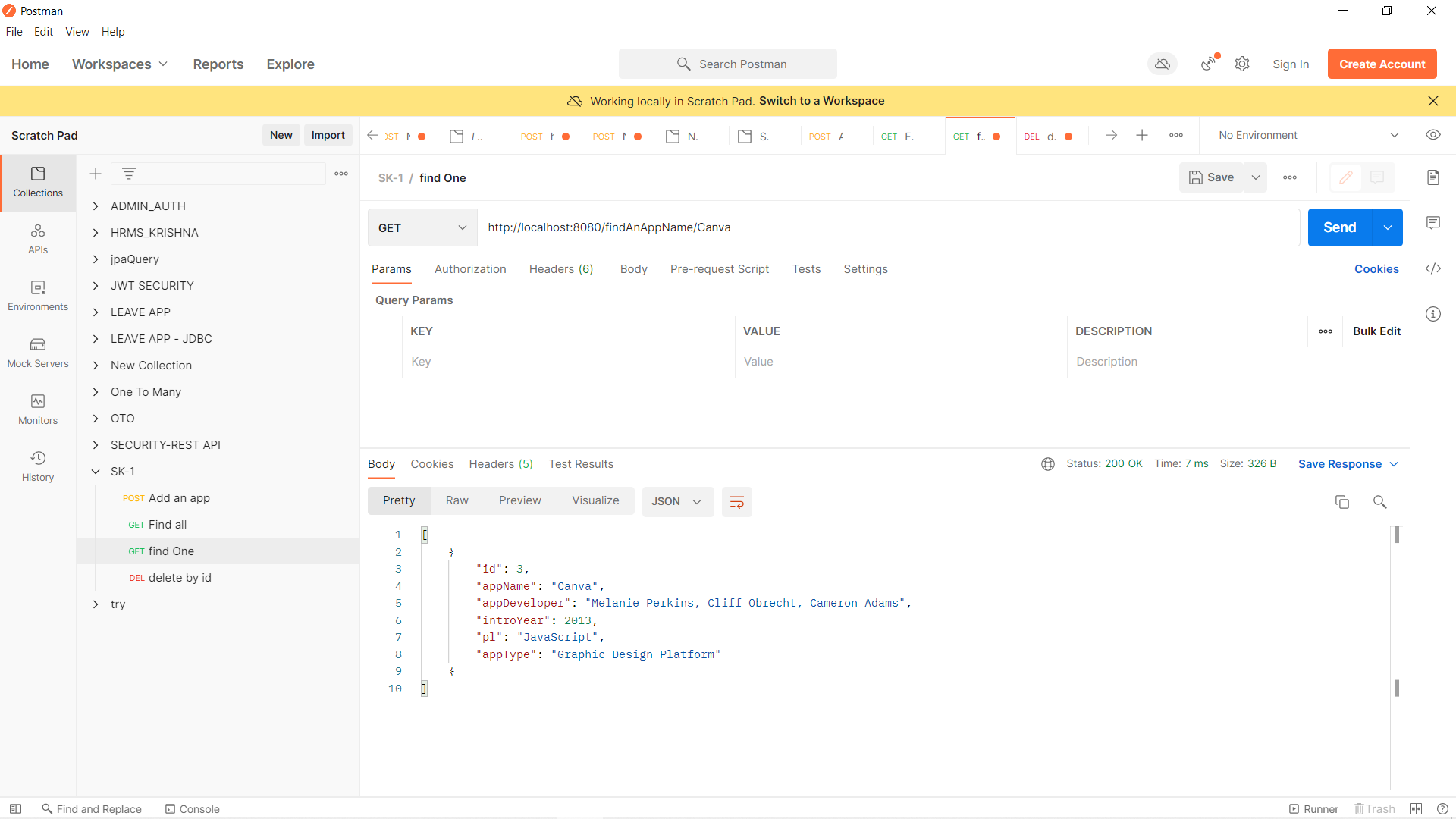
**Method: GET**

**Url:** [**http://localhost:8080/findAnApp/Canva**](http://localhost:8080/findAnApp/Canva4)

**OUTPUT:**

**Http Status: 200 OK**

I have passed appName: Canva as the parameter in the url. The data exists with that name and those details are displayed in the **Body section** of the **Response.**



1. **Find by Name, if no app found:**

**INPUT:**

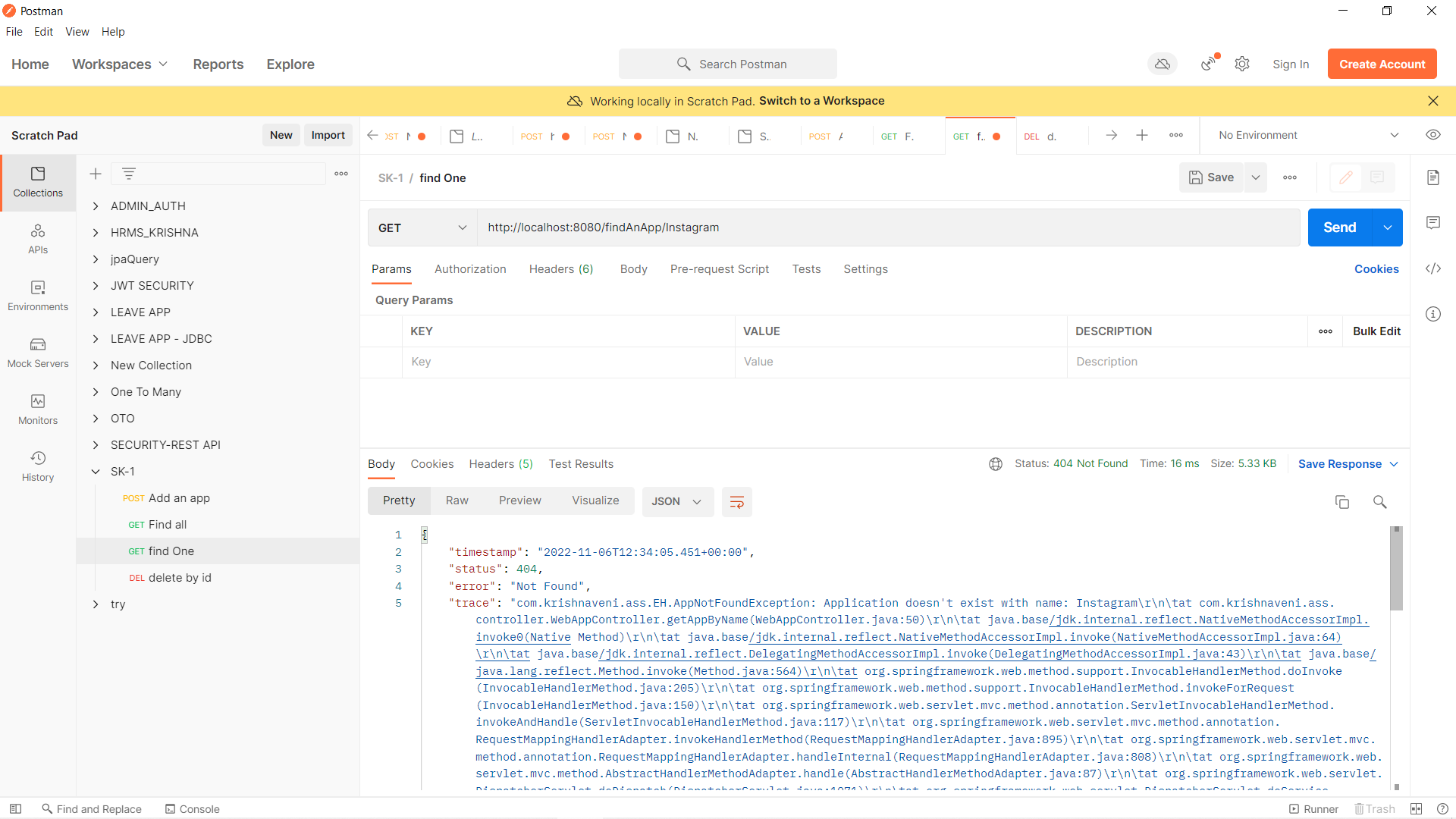
**Method: GET**

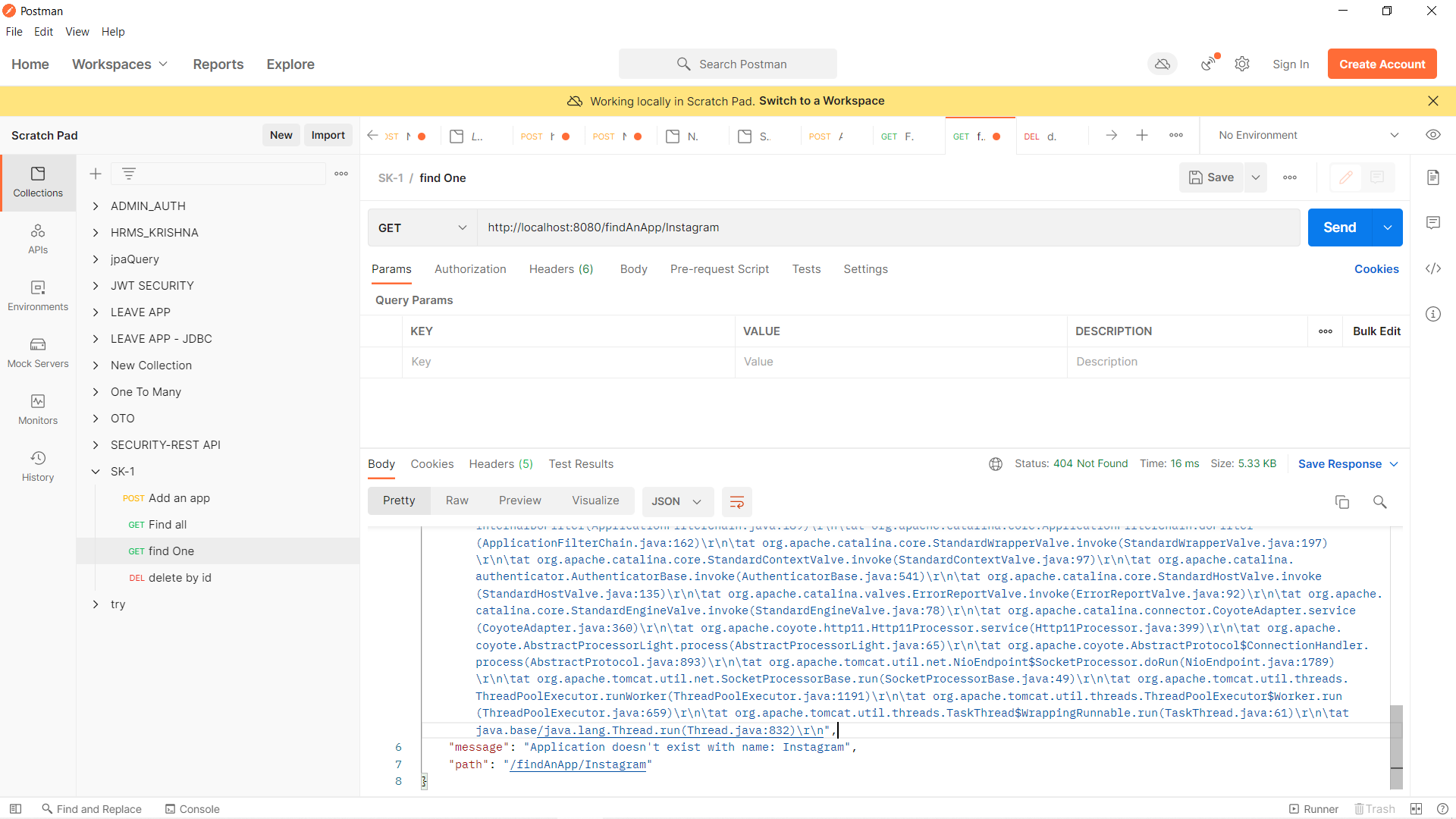
**Url:** [**http://localhost:8080/findAnApp/Instagram**](http://localhost:8080/findAnApp/Instagram)

**OUTPUT:**

**Http Status: 404 Not Found**

The data with appName: Instagram is not found. So, **AppNotFoundException** details are displayed.





1. **Delete by “id”:**

**INPUT:**

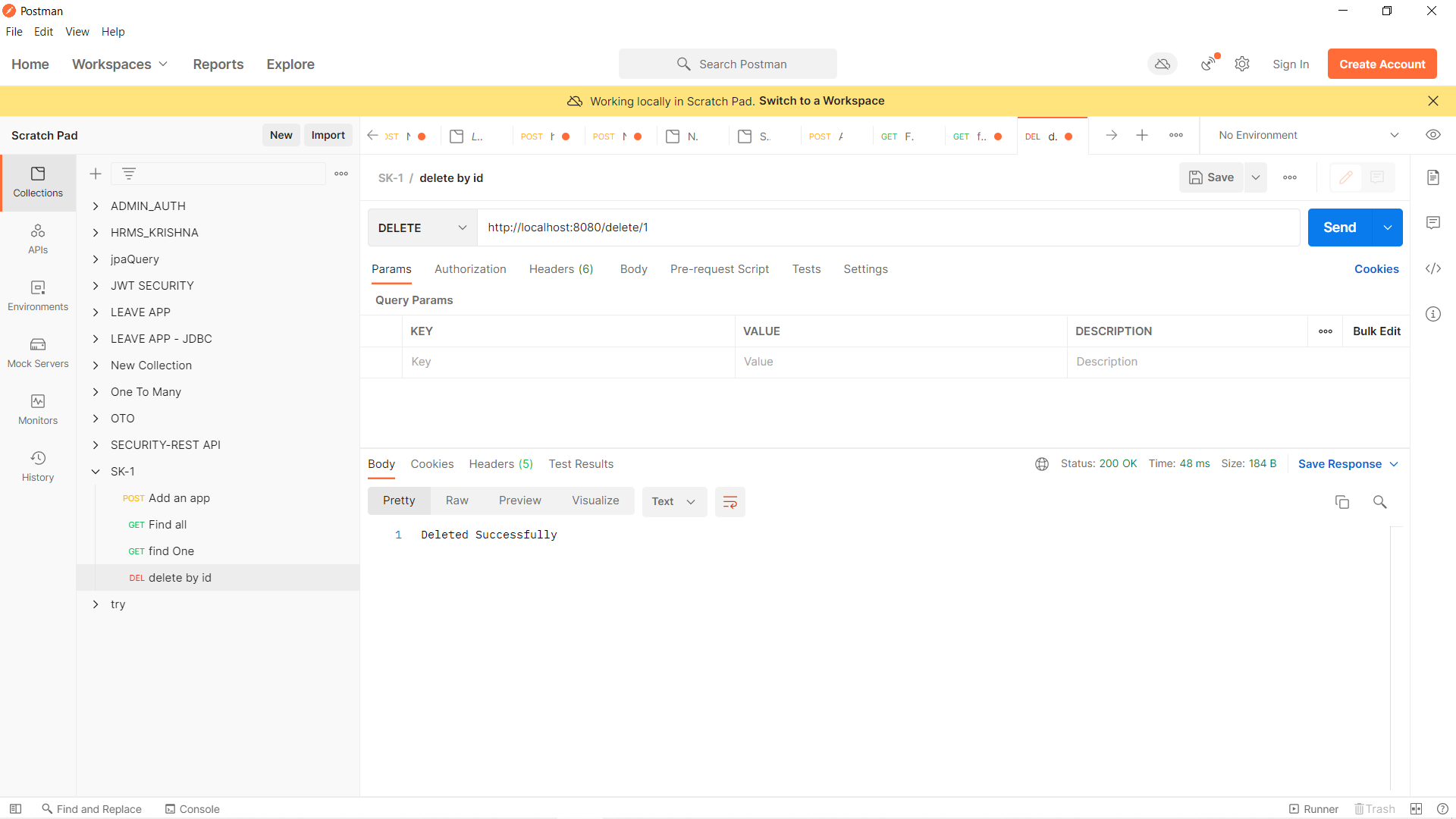
**Method: DELETE**

**Url:** [**http://localhost:8080/delete/1**](http://localhost:8080/delete/1)

**OUTPUT:**

**Http Status: 200 OK**

The app with **id:1** is found in the database. So, it is deleted successfully.



**Checking the database whether the data has been removed or not:**

