Getting data from REST

Theory | Practice | 9% completed, 0 problems solved

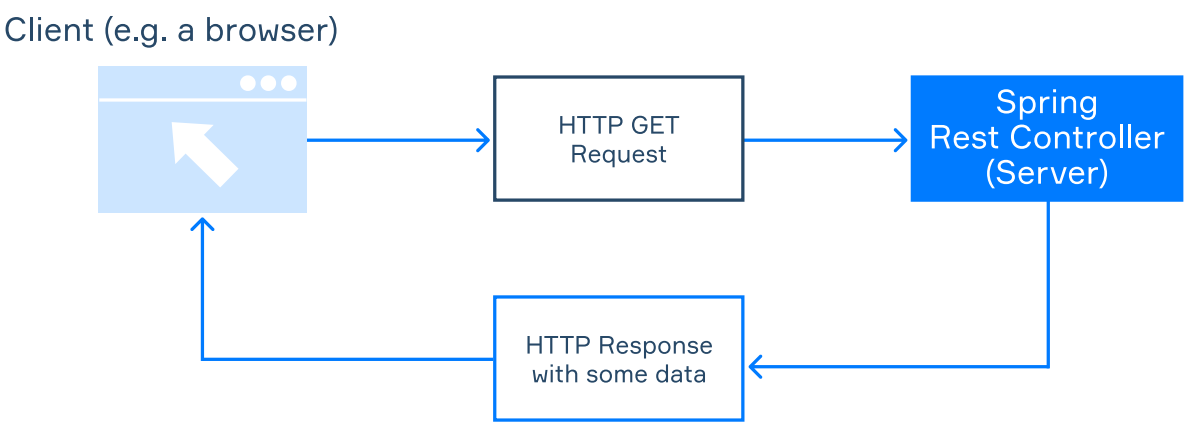
Theory

22 minutes reading

Verify to skip | Start practicing

Web-based applications communicate with a server via **API** — various methods that can be processed through **HTTP** (HyperText Transfer Protocol) **requests**. A **controller** is a part of the application that handles these API methods.

In this topic, we will take a look at how you can implement a basic **REST-based controller** for retrieving data through `GET` requests. The diagram below outlines the typical flow of a REST API when a `GET` request is sent to the server through Spring.



§1. Rest Controller

The `@RestController` annotation usually sits on top of the class. It makes a class provide exact endpoints (a requested URL) to access the REST methods. The class along with class methods can tell which requests suit your case. All appropriate requests will be sent to the specific method of this class.

Suppose that we want to create an API. When a user accesses a specific URL, they receive `1`. To make it possible with Spring, we will implement two annotations. The first annotation is `@RestController` that is used to handle any REST requests sent by a user to the application. To create a `@RestController`, we should create a class and annotate it with `@RestController`:

▼ Java

```java
import org.springframework.web.bind.annotation.*;

@RestController
public class TaskController {

}
```

▼ Kotlin

```kotlin
@RestController
class TaskController {

}
```

The `@RestController` annotation is a wrapper of two different annotations:

5 required topics

- Basic project structure
- HTTP messages
- Introduction to Spring Web MVC
- REST
- Postman

1 dependent topic

- Posting and deleting data via REST

1. `@Controller` contains handler methods for various requests. Since we opt for `@RestController`, the methods are related to REST requests.
2. `@ResponseBody` contains an object of each handler method. They will be represented in JSON format. When we send a request, the response we receive is in JSON format. This will become clear when we start working with objects in our `GET` requests.

We can implement methods to handle various REST requests in `@RestController`. To implement a `GET` request, we can use a `@GetMapping` annotation. It indicates what URL path should be associated with a `GET` request. After that, we can implement a method that is executed when the `GET` request is received at that path. For example, we can create a `GET` request that returns `1` when *http://localhost:8080/test* is accessed:

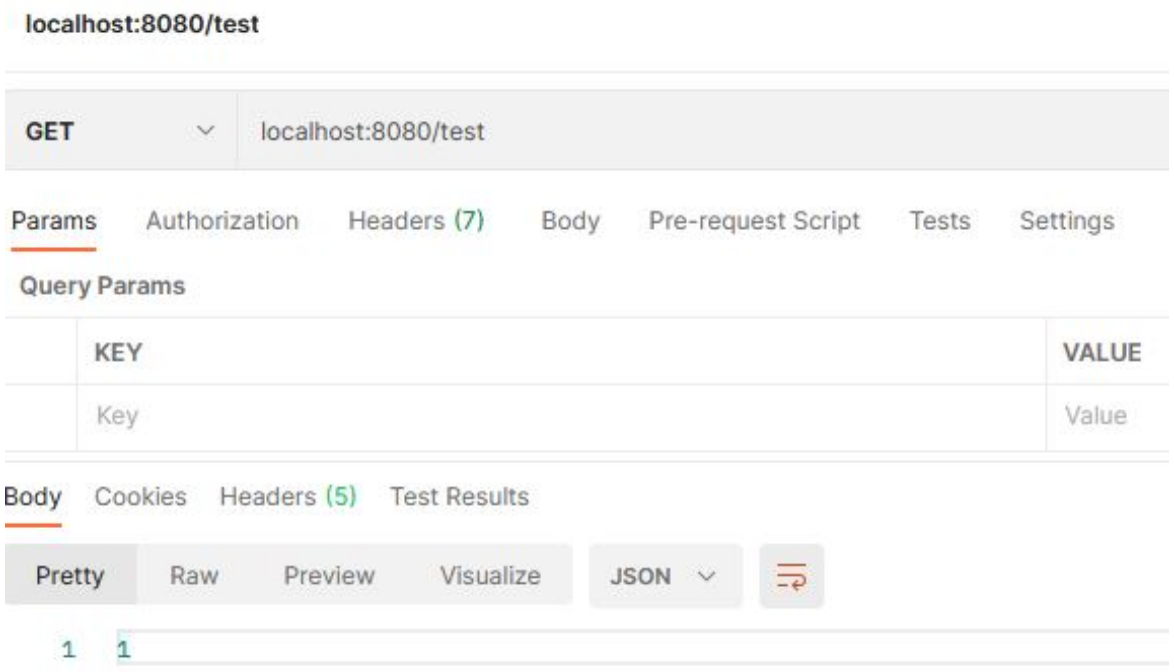
▼ Java

```java
@RestController
public class TaskController {

    @GetMapping("/test")
    public int returnOne() {
        return 1;
    }
}
```

▼ Kotlin

```kotlin
@RestController
class TaskController {
    @GetMapping("/test")
    fun returnOne(): Int {
        return 1
    }
}
```

When you send a request to `http://localhost:8080/test`, you will receive `1` in return.



In addition to Postman, it is also possible to send a `GET` request to the server through a browser. To do so, simply open the browser, and navigate to the same URL as Postman (in this example, *http://localhost:8080/test*).

§2. GET with Collections

A list is a good way to store data. Sometimes, we want to return a full list or a specific list index when a `GET` request is received. We can adjust our `@GetMapping` annotation to do so.

We need to create a simple object to store in our list. Call it `Task`. It will implement a **basic constructor** as well as **getters** and **setters** for each of the object properties:

▼ Java

```
public class Task {

    private int id;
    private String name;
    private String description;
    private boolean completed;

    public Task() {}


    public Task(int id, String name, String description, boolean
completed) {

        this.id = id;

        this.name = name;

        this.description = description;

        this.completed = completed;

    }


    // getters and setters

}
```

▼ Kotlin

```
class Task(
    var id: Int,
    var name: String,
    var description: String,
    var completed: Boolean
) {

}
```

It is very important to implement getters and setters. If they are not implemented, Spring will not be able to display object contents correctly. Spring will try to return all data from our controller in JSON format or similar. To construct a representation of our object that can be read properly, Spring needs getters and setters to access the object properties.

After that, we can implement a collection to store our tasks. We are going to use a list. When we work with Spring, we can end up facing a lot of `GET` requests at the same time. In this case, it would be a good idea to use an immutable collection to eliminate any thread-based issues. We also need to make sure that our collection can be used by our application:

▼ Java

```java
@RestController
public class TaskController {
    private final List<Task> taskList = List.of(
            new Task(1, "task1", "A first test task", false),
            new Task(2, "task2", "A second test task", true)
    );
}
```

▼ Kotlin

```kotlin
@RestController
class TaskController {
    val taskList = listOf(
        Task(1, "task1", "A first test task", false),
        Task(2, "task2", "A second test task", true)
    )
}
```

In the snippet above, we have created the `Task` list and populated it with sample tasks. You can start working with the object from a database query right away. After that, we need to create a `@GetMapping` function that can be used to retrieve data from the tasks collection.

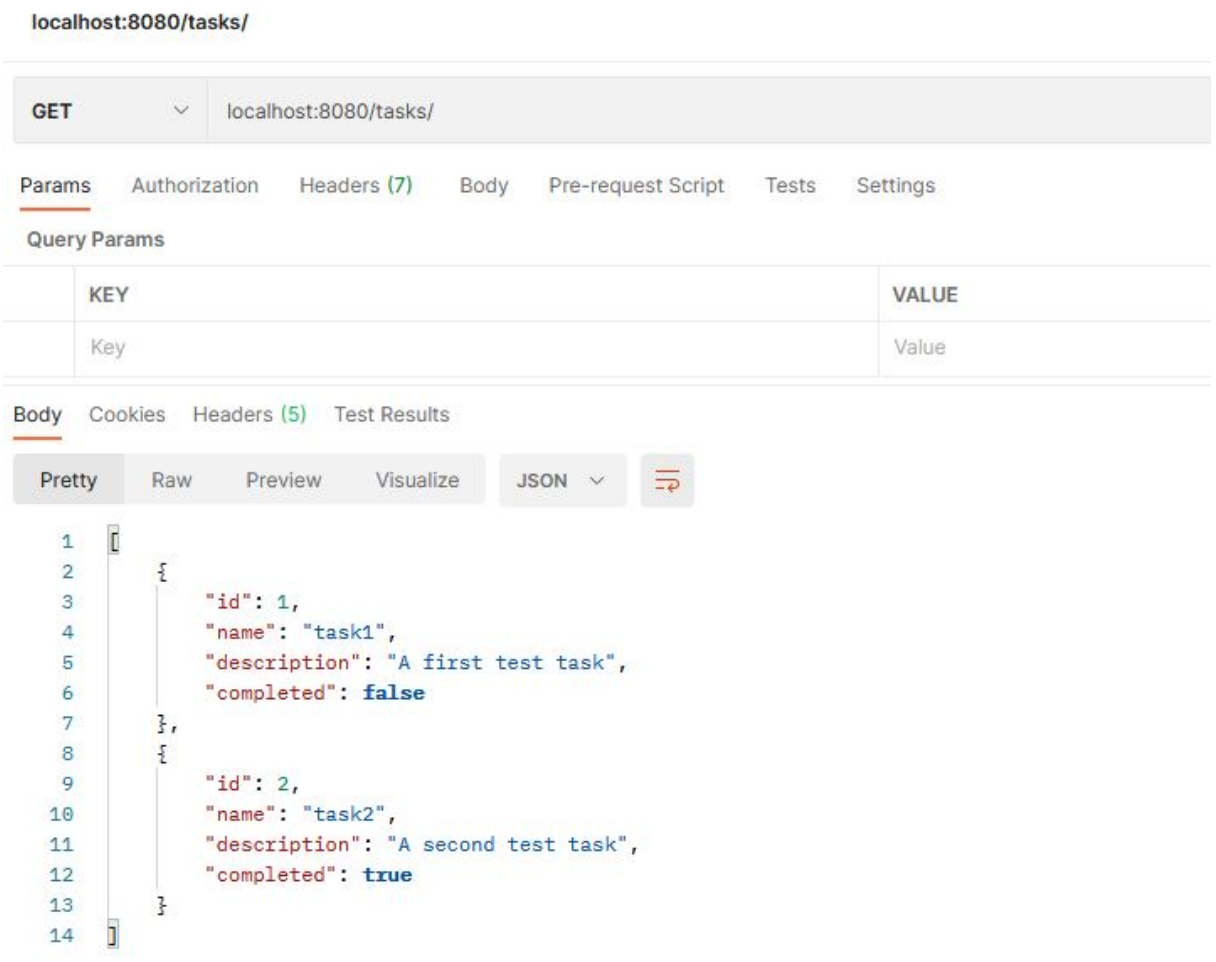
▼ Java

```java
@RestController
public class TaskController {
    private final List<Task> taskList = List.of(
            new Task(1, "task1", "A first test task", false),
            new Task(2, "task2", "A second test task", true)
    );

    @GetMapping("/tasks")
    public List<Task> getTasks() {

        return taskList;

    }

}
```

▼ Kotlin

```kotlin
@RestController
class TaskController {
    val taskList = listOf(
        Task(1, "task1", "A first test task", false),
        Task(2, "task2", "A second test task", true)
    )

    @GetMapping("/tasks")
    fun getTasks(): List<Task> {

        return taskList

    }

}
```

Now, when we make a request to *http://localhost:8080/tasks/*, we will see all tasks that have been added earlier:



In addition to a `List`, it is also possible to return other types of collections from a `RestController`. As in case of a list, a `Set` is converted to a JSON array. However, a `Map` is converted to a JSON key-value structure.

§3. @PathVariable

We may want to modify the code above so that users could enter an ID to specify which task they would like to retrieve. To do this, we will need to add a `@PathVariable` annotation to `@GetMapping`. The code below shows how we can add an ID to our `getTask` function:

▼ Java

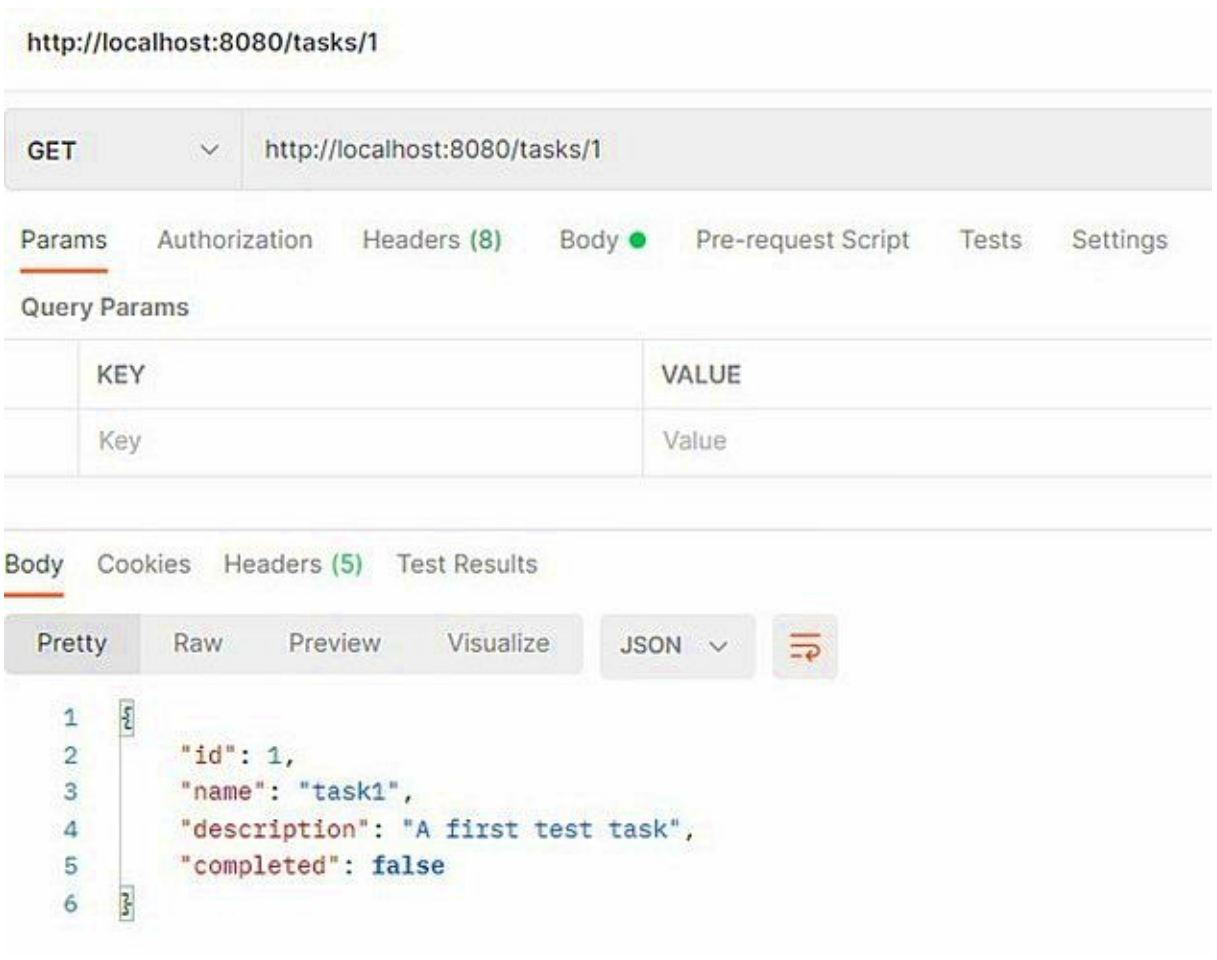
```java
@RestController
public class TaskController {
    private final List<Task> taskList = List.of(
        new Task(1, "task1", "A first test task", false),
        new Task(2, "task2", "A second test task", true)
    );

    @GetMapping("/tasks/{id}")
    public Task getTask(@PathVariable int id) {

        return taskList.get(id - 1); // list indices start from 0

    }

}
```

▼ Kotlin

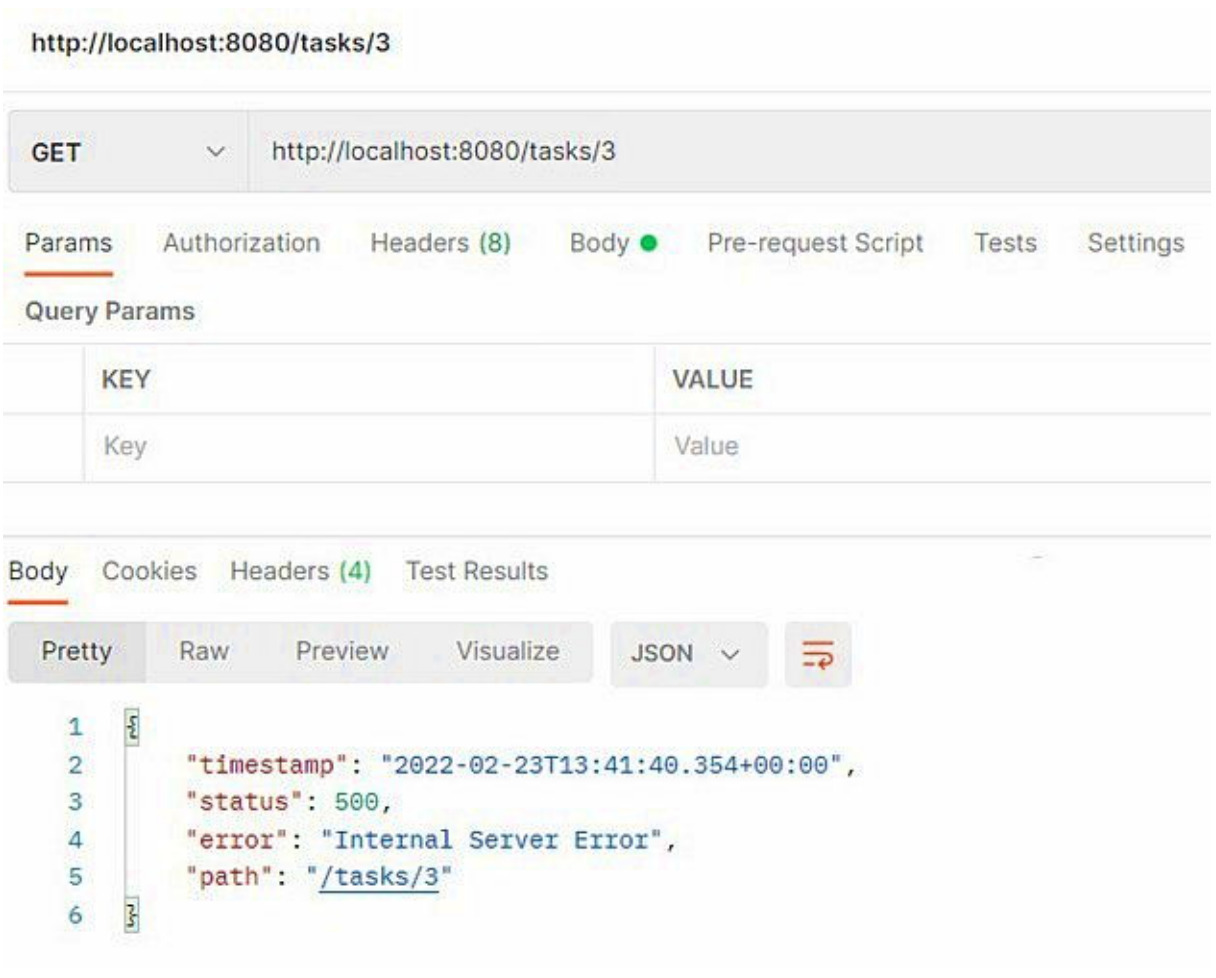
```kotlin
@RestController
class TaskController {
    val taskList = listOf(
        Task(1, "task1", "A first test task", false),
        Task(2, "task2", "A second test task", true)
    )

    @GetMapping("/tasks/{id}")
    fun getTask(@PathVariable id: Int): Task? {

        return taskList[id - 1] // list indices start from 0
```

```
    }
}
```

We added `{id}` to the `@GetMapping` annotation to tell Spring that we expect the `id` parameter. We can place the `id` variable as `@PathVariable` in the arguments of our `getTask` method. It indicates to Spring how to map the parameter provided in `@GetMapping` to the function. After that, the function will return only one element rather than the whole collection. A request to *http://localhost:8080/tasks/1* gives us the first task in the list:



However, if we provide an invalid id, the `get` method will throw an exception and we will receive a **500 error**, similar to what is pictured below:



§4. Customizing the status code

By default, a method annotated with `@GetMapping` returns the status code `200 OK` in the response if a request has been processed successfully and the status code `500` if there is an uncaught exception. However, we can change this default status code by returning an object of the `ResponseEntity<T>` class as the result.

There is an example below when we return `202 ACCEPTED` instead of `200 OK`.

▼ Java

```
@GetMapping("/tasks/{id}")
public ResponseEntity<Task> getTasks(@PathVariable int id) {
    return new ResponseEntity<>(taskList.get(id - 1),
HttpStatus.ACCEPTED);
}
```

▼ Kotlin

```
@GetMapping("/tasks/{id}")
fun getTasks(@PathVariable id: Int): ResponseEntity<Task?>? {
    return ResponseEntity(taskList[id - 1], HttpStatus.ACCEPTED)
}
```

Actually, the status code `202 ACCEPTED` is not the best example for this case, but it clearly demonstrates the possibility to change the status code.

§5. Conclusion

A controller is the first component that meets and greets a web request. In this topic, we have covered how to define a `GET` method in a `@RestController` annotated class to receive data from a web application. This request type is common in APIs and is often required to return sets or single elements.

On the one hand, web-app developers need to keep the handlers short and clear as it helps to find the right handler and create a correct request quickly. On the other hand, web apps are clients for other web apps. It means that they can call controllers of other applications. That's why you also need to know foreign handlers to figure out what requests they can handle.

Report a typo

243 users liked this piece of theory. **6** didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing Verify to skip

Table of contents:

Comments (32) Useful links (5) Show discussion