Posting and deleting data via REST

Theory

16 minutes reading

When users receive data from web applications, they may want to **add** new or **delete** the existing data. With `POST` requests users can add new information by sending **values** they want to upload. A `DELETE` request allows users to remove the existing data from an application. When users send `POST` or `DELETE` requests, they are processed by the `@RestController`. The controller takes the appropriate actions depending on the method. In this topic, we will learn how to implement `POST` and `DELETE` methods via **Spring**.

We will use the REST Resource Naming Guide throughout this topic; it governs standard naming conventions. Use this site if you want to learn more about API namings.

1 required topic
Getting data from REST
4 dependent topics
Handling requests with bodies
Exception handling
Bean validation
Custom User Store

§1. @PostMapping

Suppose you want to create an application where users can add names and addresses of the people they know. To add a person to the address book, a user needs to send the data to the server, while the server needs to store it somewhere. To make it possible, implement `@PostMapping` in the `@RestController`.

> We advise you to use a **thread-safe object** to work with data in `@RestController`. The controller can get multiple requests at the same time, and the requests are processed by different threads. If the object is not thread-safe, multiple requests can lead to data loss and other unexpected errors when data is processed with `POST` or `DELETE` requests

In our example, we want to store mappings from people to addresses, so use a `Map` object. We can use `ConcurrentHashMap` to implement a thread-safe `Map` in our application:

▼ Java

```
@RestController
public class AddressController {
    private ConcurrentMap<String, String> addressBook = new ConcurrentHashMap<>();
}
```

▼ Kotlin

```
@RestController
class AddressController {
    private val addressBook = ConcurrentHashMap<String, String>()
}
```

With `ConcurrentHashMap` we can set up a `@PostMapping` that takes a person's name and address and adds them to the `Map`. Since the user wants to send data with a `POST` request, we need to use a `@RequestParam` to send the data with a `POST` request.

`@RequestParam` is a **variable** provided by a user in the **query parameters**. It is used during handling of `POST` requests. `@RequestParam` can be provided in two ways:

1. In the query parameters section of a REST request. In Postman, it can be found in the **Params** section, labeled as **Query Params**;
2. In the URL path, in the following format: `localhost:<port>/<ApiPath>?<Param>=<value>&<Param>=<value>`.

In the examples below, the Spring port is set to 8080, so all `POST` and `DELETE` requests are made at `localhost:8080`.

When we provide a parameter through the query parameters, we need to set a name and a value. The name of the parameter should match the name of the `@RequestParam`, and the value should be the same type as the `@RequestParam` variable. The following code is an example of how `@RequestParam` can be used with `@PostMapping` to add the data to the address book:

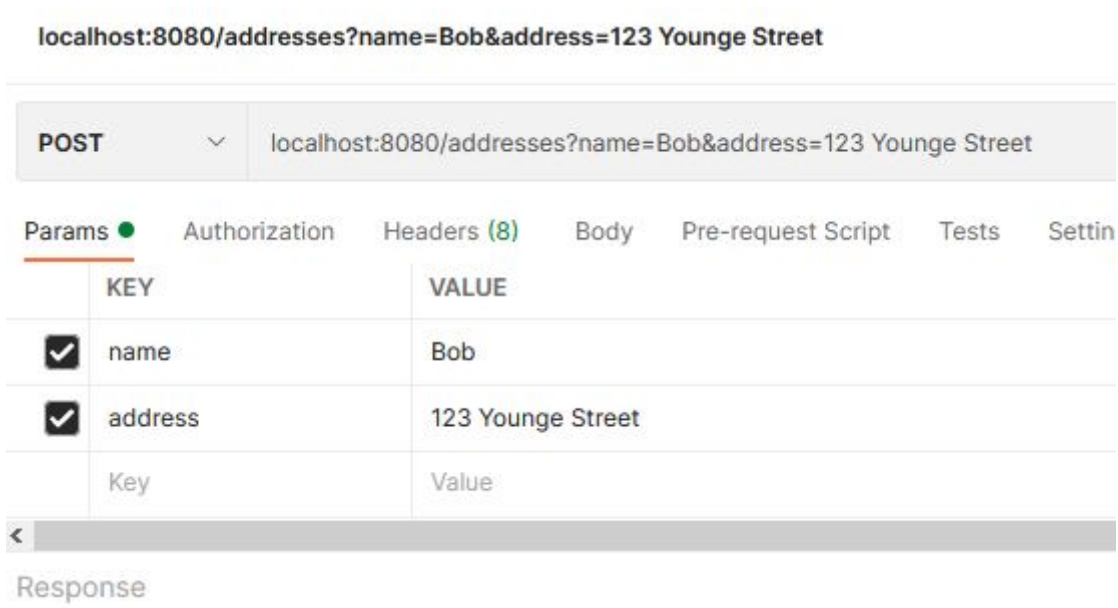
▼ Java

```java
@RestController
public class AddressController {
    private ConcurrentMap<String, String> addressBook = new ConcurrentHashMap<>();

    @PostMapping("/addresses")
    public void postAddress(@RequestParam String name, @RequestParam String address) {
        addressBook.put(name, address);
    }
}
```

▼ Kotlin

```kotlin
@RestController
class AddressController {
    private val addressBook = ConcurrentHashMap<String, String>()

    @PostMapping("/addresses")
    fun postAddress(@RequestParam name: String, @RequestParam address: String) {
        addressBook[name] = address
    }
}
```

In this `@PostMapping`, we expect two `@RequestParam` with a request. The first is the name of the `String` type. The second is the address, also of the `String` type. When users send a `POST` request to the `/addresses` path, they provide two parameters in the request body. When the request is sent, the name and address are added to `ConcurrentHashMap`.



To test whether the `POST` was a success, you can implement a `GET` request that returns a requested address based on the provided name:

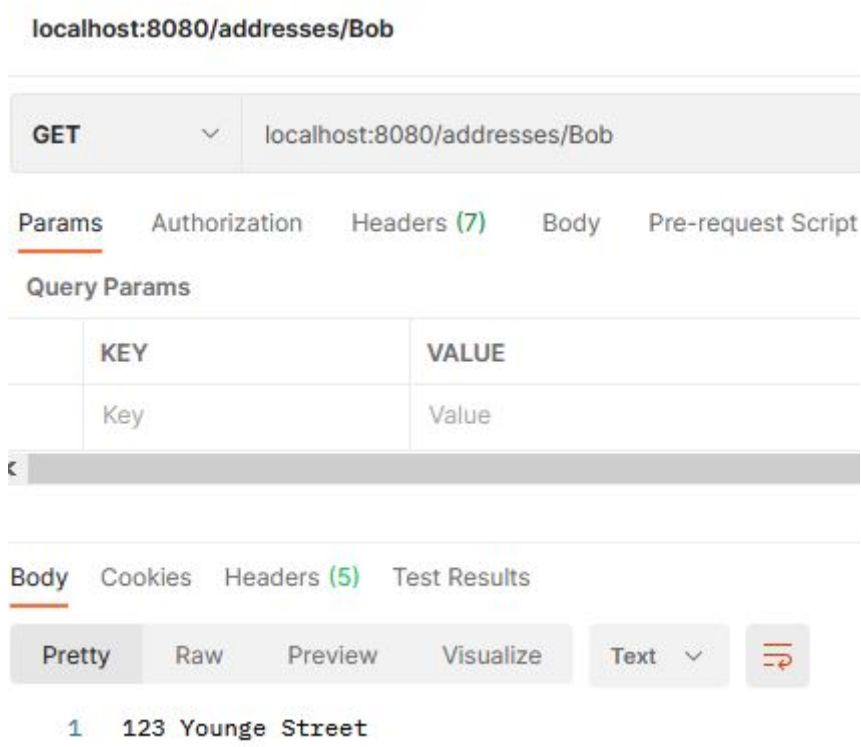
▼ Java

```java
@RestController
public class AddressController {
    private ConcurrentMap<String, String> addressBook = new ConcurrentHashMap<>();

    @PostMapping("/addresses")
    public void postAddress(@RequestParam String name, @RequestParam String address) {
        addressBook.put(name, address);
    }


    @GetMapping("/addresses/{name}")

    public String getAddress(@PathVariable String name) {

        return addressBook.get(name);

    }

}
```

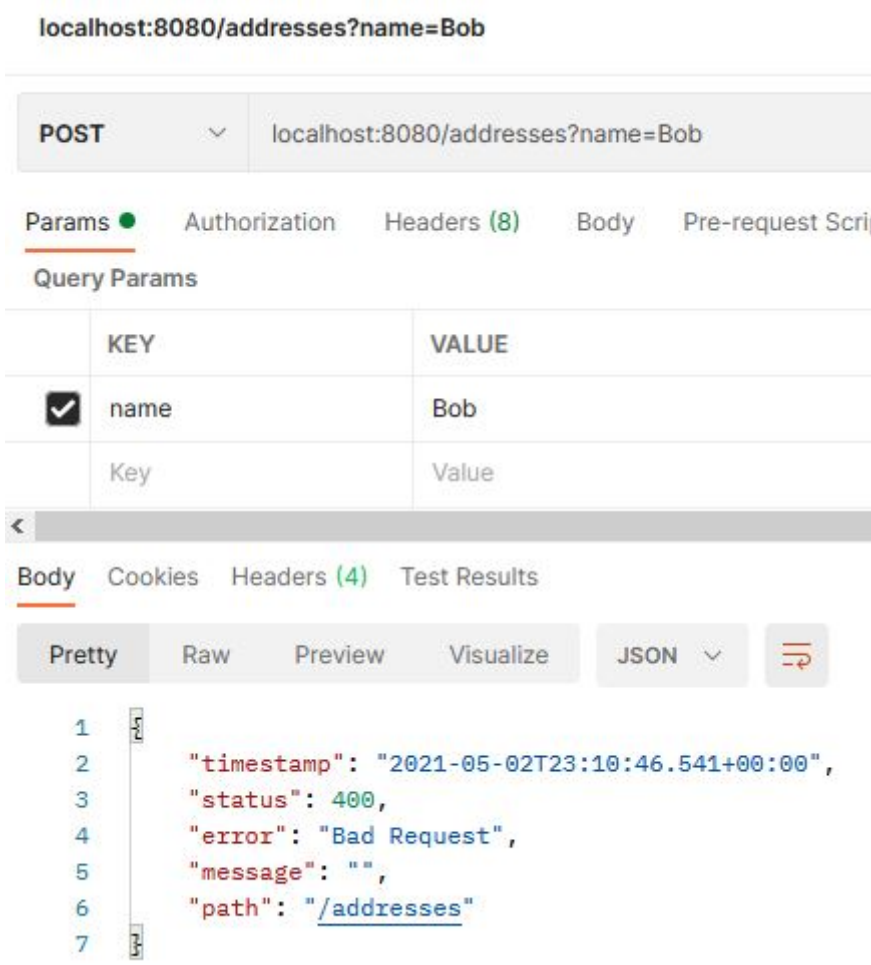
▼ Kotlin

```kotlin
@RestController
class AddressController {
    private val addressBook = ConcurrentHashMap<String, String>()

    @PostMapping("/addresses")
    fun postAddress(@RequestParam name: String, @RequestParam address: String) {
        addressBook[name] = address
    }


    @GetMapping("/addresses/{name}")

    fun getAddress(@PathVariable name: String): String? {

        return addressBook[name]

    }

}
```

In the previous `POST` request, we have added `Bob` that is mapped to `123 Younge Street`. Now, if we send a request to `/addresses/Bob`, we expect to get `123 Younge Street` as shown below:



But what happens when a parameter is either missing or invalid? A user will receive `400 Bad Request` as shown below:



If this error occurs, reexamine the parameters to make sure that they are correct.

§2. @DeleteMapping

Apart from adding new data, sometimes users need to delete some data too. In our address book, we may want to delete a name if it is no longer required. In this situation, we can use `@DeleteMapping` to send a request to delete some portion of our data.

Using `@RequestParam` we can pass a parameter to the `@DeleteMapping` handler. The parameter that needs to be passed in our example is the name of the person we want to delete. Once the name has been provided, we can remove the value from the `Map` and return a message to indicate that it has been successfully deleted:

▼ Java

```
@RestController
public class AddressController {
    private ConcurrentMap<String, String> addressBook = new ConcurrentHashMap<>();

```

```
    @DeleteMapping("/addresses")
    public String removeAddress(@RequestParam String name) {
        addressBook.remove(name);
        return name + " removed from address book!";
    }

}
```

▼ Kotlin

```
@RestController
class AddressController {
    private val addressBook = ConcurrentHashMap<String, String>()

    @DeleteMapping("/addresses")
    fun removeAddress(@RequestParam name: String): String {
        addressBook.remove(name)
        return "$name removed from address book!"
    }

}
```

To verify that the mapping has been removed, we can send a `GET` to return the contents of the `addressBook` variable. Take a look at the snippet below. It shows the whole controller:

▼ Java

```
@RestController
public class AddressController {
    private ConcurrentMap<String, String> addressBook = new ConcurrentHashMap<>();

    @PostMapping("/addresses")
    public void postAddress(@RequestParam String name, @RequestParam String address) {
        addressBook.put(name, address);
    }


    @GetMapping("/addresses")

    public ConcurrentMap<String, String> getAddressBook() {

        return addressBook;

    }



    @DeleteMapping("/addresses")

    public String removeAddress(@RequestParam String name) {

        addressBook.remove(name);

        return name + " removed from address book!";

    }

}
```

▼ Kotlin

```
@RestController
class AddressController {
    val addressBook = ConcurrentHashMap<String, String>()

    @PostMapping("/addresses")
```

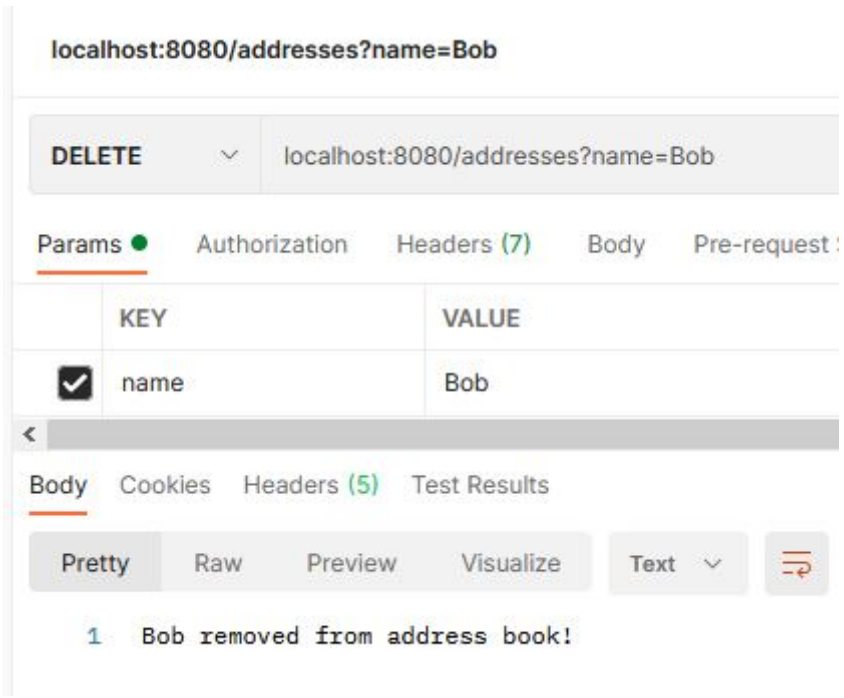
```
    fun postAddress(@RequestParam name: String, @RequestParam address: String) {
        addressBook[name] = address
    }


    @GetMapping("/addresses")

    fun getAddressBook(): ConcurrentMap<String, String> {

        return addressBook

    }



    @DeleteMapping("/addresses")

    fun removeAddress(@RequestParam name: String): String {

        addressBook.remove(name)

        return "$name removed from address book!"

    }

}
```

Once `@DeleteMapping` has been established, we only need to send a `DELETE` request to the `/addresses` URL with the address we want to delete in the query parameters. To test this, let's first populate our `Map` with data. To do this, we can send a few `POST` requests to the web application. Consider the following two `POST` requests:

- `localhost:8080/addresses?name=Bob&address=123 Younge Street`
- `localhost:8080/addresses?name=Alice&address=200 Rideau Street`

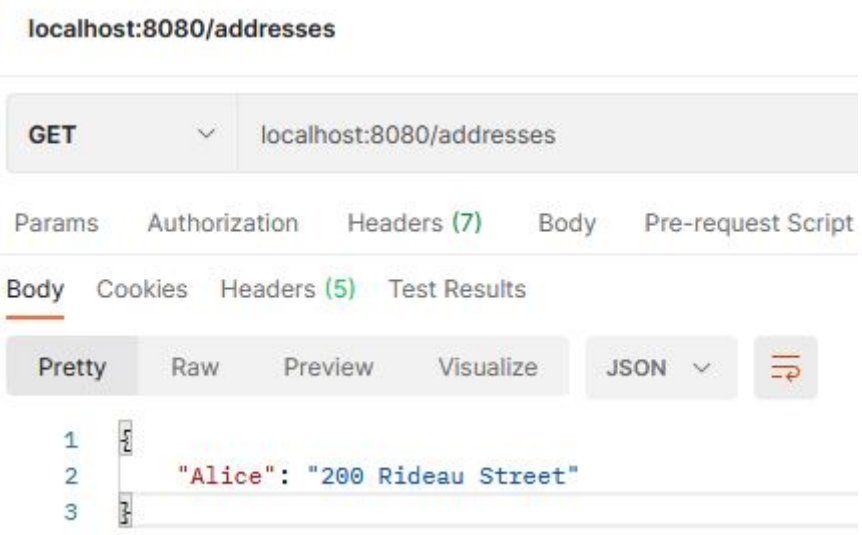
This will add two entries to the `Map`, the first is `Bob` living on `123 Younge Street`. The second is `Alice` living on `200 Rideau Street`. We can verify whether the entries were added with a `GET` request to `/addresses`.

Now, suppose that we want to delete the entry associated with `Bob`. We need to send a `DELETE` request to the `/addresses` mapping, passing the name parameter with the `Bob` value.



Once the data has been removed, we can verify that the request has been completed successfully by sending another `GET` request for the whole `Map`. As a result, the value for Bob is removed from the `Map`:

Table of contents:



§3. Conclusion

In this topic, we have discussed how we can add and remove data with `POST` and `DELETE` requests. With `@RequestParam` annotation, it is possible to send data through the query parameters, rather than through the path as with `@PathVariable`. When we work with the stored data in a `@RestController`, it is important to remember that the application can process multiple requests at once. So, it is essential to implement thread-safe objects. They ensure that no thread-related data errors occur. When you work with `@RequestParam`, remember that the `400 Bad Request` error will occur if parameters are missing or incorrect. Review the parameters if you happen to see this error. Make sure that none of them are incorrect or missing. This will help you with building complex but steady REST APIs that can handle user input.

Report a typo

181 users liked this piece of theory. 4 didn't like it. **What about you?**

Start practicing

Unskip this topic

Comments (14) Useful links (0) Show discussion