


# Handling requests with bodies

Theory

Practice

 16% completed, 0 problems solved

▼

## Theory

🕒 12 minutes reading

Skip this topic

Start practicing

When we want to pass information to a web application server, we usually opt for `POST` methods. We can actually send information of any type, even an unformatted raw string. But we tend to work with JSON, as it is one of the easiest formats to work with. JSON data can be easily converted to objects with Spring, so we can work with more complex data.

We will use an annotation called `@RequestBody` to accept JSON data via `@RestController`. This annotation can send data of a specific format, including JSON, in a request's body.

We will use the basic project. If you don't have this application, visit the [Spring Initializr](#) site and generate it with Gradle and Java.

## §1. Sending an object to the server

The `@RequestBody` annotation is used in `@RestController` to send data to a path through the **body** of the request. A request body can be used to send data in a variety of formats. By default, Spring expects data in **JSON format**, so we will start by looking at how JSON data can be sent using a `@RequestBody` annotation.

First, we will create an object to represent the JSON data that will be sent to the application. In this example, we will create an application that works with user data, including the name, id, phone number, and status of the user's account. Our object can be set up as shown below:

▼ Java


```
1 public class UserInfo {
2
3     private int id;
4     private String name;
5     private String phone;
6     private boolean enabled;
7
8     // getters and setters
9
10    UserInfo() {}
11
12 }
```


▼ Kotlin

```
1 class UserInfo(
2     val id: Int? = null,
3     val phone: String? = null,
4     val enabled: Boolean? = null,
5     val name: String? = null
6 )
```

Now, let's create a simple `POST` request that accepts a JSON representation of the `UserInfo` object. The `POST` request will return the user's account name and the

### 1 required topic






[Posting and deleting data via REST](#)

▼

### 2 dependent topics

[Application layers](#)

▼



[POST vs. PUT requests](#)

▼

status of the account:

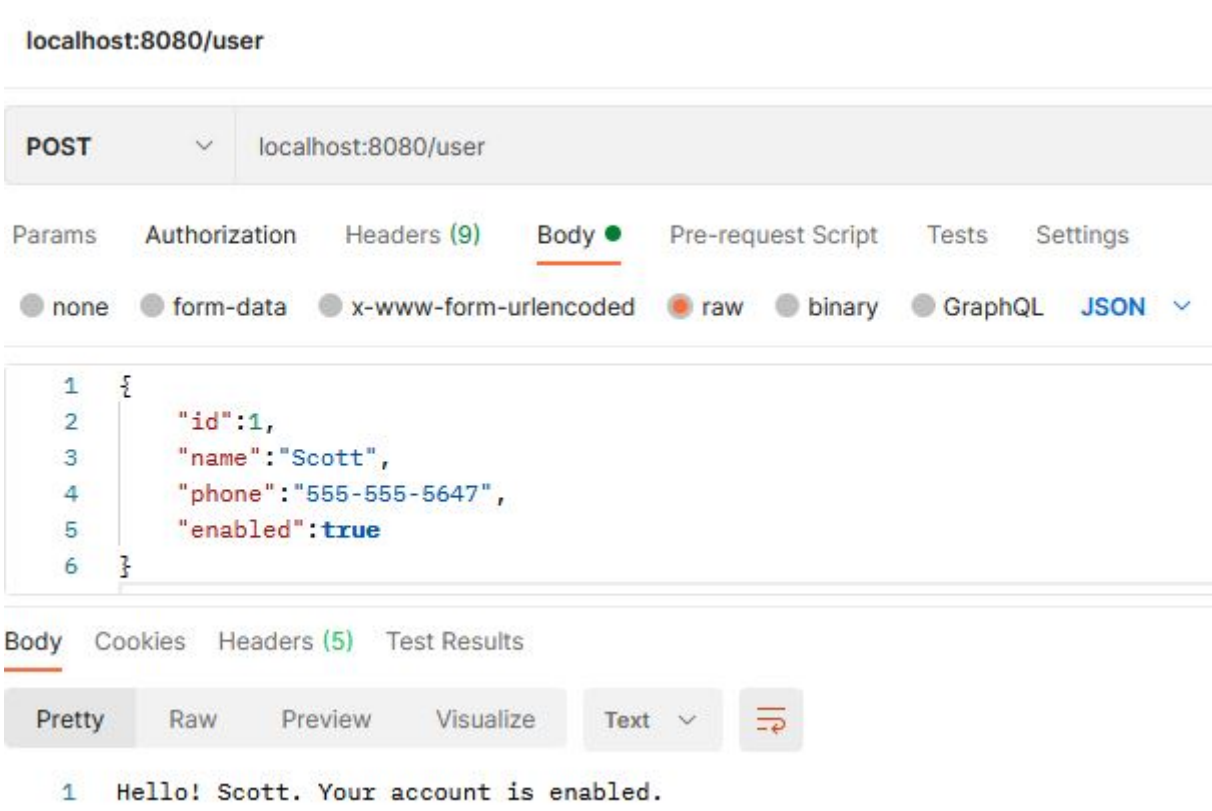
#### ▼ Java

```
1  @RestController
2  public class UserInfoController {
3
4      @PostMapping("/user")
5      public String userStatus(@RequestBody UserInfo user) {
6          if (user.isEnabled()) {
7              return String.format("Hello! %s. Your account is
enabled.", user.getName());
8
9          } else {
10
11              return String.format(
12
13                  "Hello! Nice to see you, %s! Your account is
disabled",
14
15                  user.getName()
16
17              );
18          }
19      }
20  }
```

#### ▼ Kotlin

```
1  @RestController
2  class UserInfoController {
3      @PostMapping("/user")
4      fun userStatus(@RequestBody user: UserInfo): String {
5          return if (user.isEnabled()) {
6              String.format("Hello! %s. Your account is enabled.",
user.name)
7          } else {
8              String.format(
9                  "Hello! Nice to see you, %s! Your account is
disabled",
10
11                  user.name
12
13              )
14          }
15      }
16  }
```

When we generate our `POST` request to the `/user` path, we need to provide a **query body** that defines a valid `UserInfo` object. This data is provided so that each object property is defined as **an entry to the JSON object**. This means that the request should contain an id, name, phone, and status:



When adding a body in Postman, you can choose the JSON format by setting the raw option and navigating to JSON in the dropdown menu. It sends the provided JSON data in the request body to the `user` object in our code. Spring can construct a `UserInfo` object with JSON properties and initialize it with getters and setters implemented in the code.

We can send more complex data to our application with `@RequestBody`. There are a few other formats we can utilize in the `@RequestBody` annotation. We will explore them in the next section.

## §2. Sending multiple objects

It is possible to send multiple JSON objects in a single request using a list of objects in our `@RequestBody`. To implement a list-based `@RequestBody`, we need to change the type from a single object to a list of objects:

### ▼ Java

```

1 @RestController
2 public class UserInfoController {
3
4     @PostMapping("/user")
5     public String userStatus(@RequestBody List<UserInfo> userList) {
6         return String.format("Added %d users", userList.size());
7     }
8 }

```

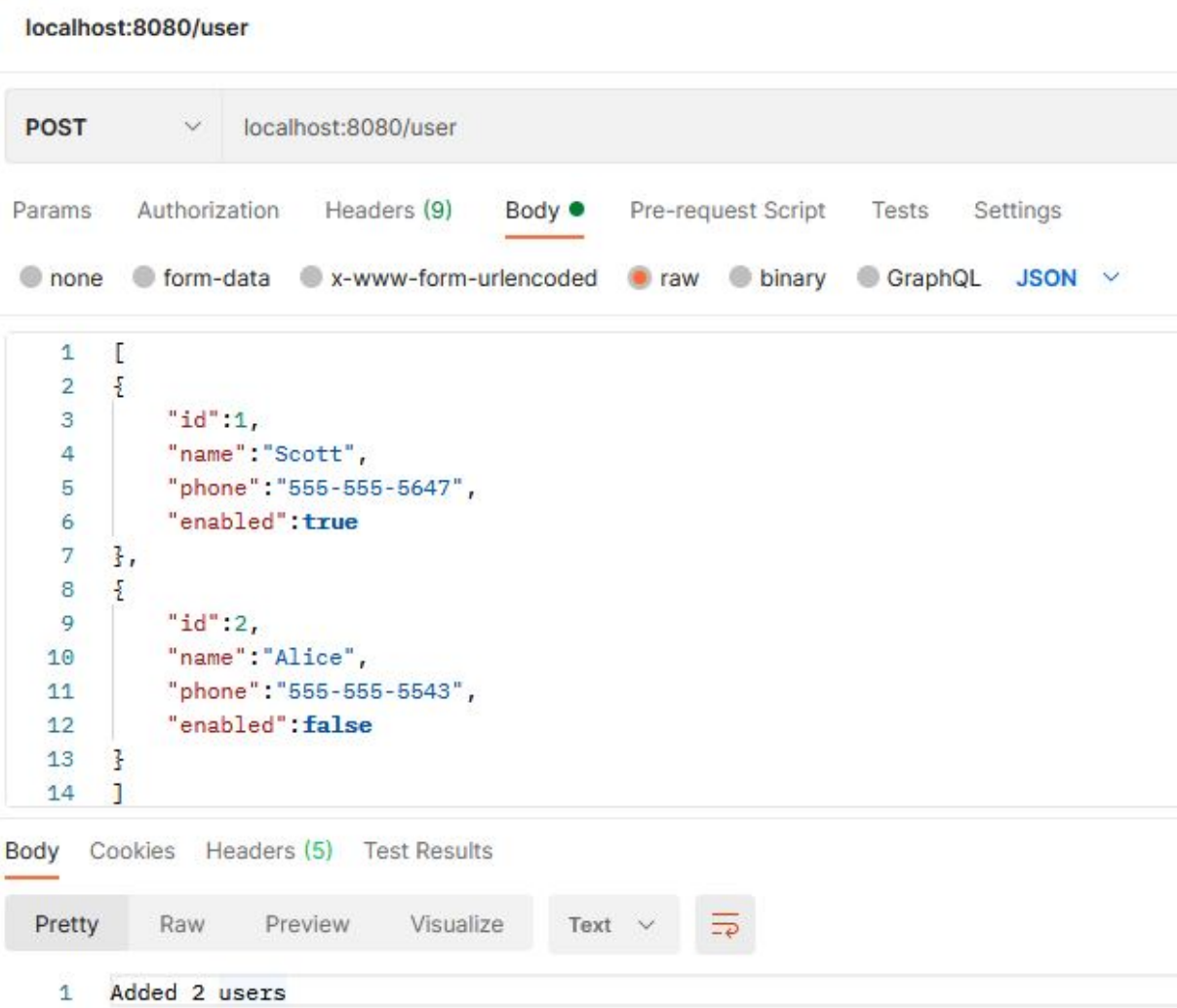
### ▼ Kotlin

```

1 @RestController
2 class UserInfoController {
3     @PostMapping("/user")
4     fun userStatus(@RequestBody userList: List<UserInfo>): String {
5         return String.format("Added %d users", userList.size)
6     }
7 }

```

In this example, our `@RequestBody` now accepts a list of `UserInfo` type. So the JSON we send to the server should now be **a list** of JSON objects. Use rectangular brackets `[]` to create a list of JSON objects. A list contains a sequence of one or more JSON objects, as shown below:



After the request has been sent, the JSON array can be iterated, and each object is placed in the `UserInfo` list. In our example above, we output the number of objects that have been passed through the request.

### §3. Additional data formats

In addition to JSON arrays, it is also possible to use a different format. For example, we can use XML to pass objects through our `@RequestBody` annotation. To do this, we need to add `consumes = MediaType.APPLICATION_XML_VALUE` to the `@PostMapping` annotation:

#### ▼ Java

```
1 @RestController
2 public class UserInfoController {
3
4     @PostMapping(value = "/user", consumes =
5     MediaType.APPLICATION_XML_VALUE)
6     public String userStatus(@RequestBody UserInfo user) {
7         return String.format("Added User %s", user);
8     }
9 }
```

#### ▼ Kotlin

```
1 @RestController
2 class UserInfoController {
3     @PostMapping(value = ["/user"], consumes =
4     [MediaType.APPLICATION_XML_VALUE])
5     fun userStatus(@RequestBody user: UserInfo): String {
6         return String.format("Added User %s", user)
7     }
8 }
```

Using the `consumes` argument, it is possible to customize the data that is sent to the `POST` request. When the `consumes` argument is added to the mapping, we also need to label the path argument with `value`. This allows Spring to distinguish between the arguments. If a `consumes` argument is not provided, it will be JSON by default. There are many other `MediaType` values. For example,

#### Table of contents:

- [1 Handling requests with bodies](#)
- [§1. Sending an object to the server](#)
- [§2. Sending multiple objects](#)
- [§3. Additional data formats](#)
- [§4. Conclusion](#)

`TEXT_PLAIN` can be used for **plain text**, and `TEXT_HTML` can be used for **HTML** formats. | [DISCUSSION](#)

A full list of available formats can be found in the [official documentation](#).

## \$4. Conclusion

In this topic, we have discussed a few ways to make `@RestController` handle requests with bodies:

1. Add a `@RequestBody` annotation to the parameter of the mapping;
2. Create a class to map a body to an object with **getters** and **setters**;
3. If we want to include a consumable that isn't JSON, add the `consumes` argument to mapping.

After these steps, your method will be able to read a passed request body in any required format.

 Report a typo

151 users liked this piece of theory. 5 didn't like it. **What about you?**



Start practicing

Skip this topic

[Comments \(12\)](#)

[Useful links \(0\)](#)

[Show discussion](#)

 JetBrains Academy

Tracks


Pricing

For organizations

About

Contribute

Careers

 Become beta tester

Be the first to see what's new

[Terms](#)

[Support](#)



Made with  by Hyperskill and JetBrains