# Performing Basic CURD Operation with Spring Boot Annotations

## Enhancing the Hello World Service with a Path Variable

The @PathVariable annotation is used to extract the value from the URI. It is most suitable for the RESTful web service where the URL contains some value. Spring MVC allows us to use multiple @PathVariable annotations in the same method. A path variable is a critical part of creating rest resources.

## Using @PathVariable

The @PathVariable annotation is used to extract data from the URL path. It allows you to define placeholders in your request mapping URL and bind those placeholders to method parameters.

We will create another hello-world-bean request with a path parameter.

*//pathvariable example  
@GetMapping*(path = "/helloworld/path-variable/{name}")  
  
*public* String helloworldwithpathvariable(*@PathVariable* String name){  
 *return* "welcome dear " +name +"to web services tutorial";  
}

To test

<http://localhost:8080/helloworld/path-variable/archana>

@ReqestParameter :

# What is @RequestParam?

The @RequestParam annotation in Spring MVC is used to extract query parameters from the request URL. Query parameters are typically appended to the URL after a question mark (?) and separated by ampersands (&). For example, in the URL http://example.com/api/products?id=123&name=Laptop, id and name are query parameters.

*//requestparameter example  
  
@GetMapping*(path ="/helloworld")  
*public* String helloworldwithrequestparams(*@RequestParam*("name") String name){  
 *return* "welcome dear " +name +"to web services tutorial request params";  
}

<http://localhost:8080/helloworld?name=archana>

# Autowiring in Springboot

@Autowired is a Spring Boot annotation that automatically injects dependencies into classes, fields, methods, and constructors:

* **How it works**

Spring Boot scans code for beans, identifies dependencies, and automatically injects the corresponding beans into the target class.

* **When to use**

@Autowired can be used to eliminate the need for manual configuration and to write decoupled code that is easier to test and maintain.

* **How to use**

@Autowired can be applied to variables, methods, and constructors. For example, to inject a UserRepository bean into a UserService, you can annotate the UserService constructor with @Autowired.

* **Types of autowiring**

There are different types of autowiring, including autowire byName and constructor injection:

* + **Autowire byName**: Uses the setter method for dependency injection. The variable name must be the same in the class where the dependency is injected and in the spring bean configuration file.
  + **Constructor injection**: Ensures that the dependency is set when the object is created and cannot be changed afterward.
* **Ambiguity**

If Spring finds multiple matching beans, it considers it an ambiguity, and you must provide additional qualifiers to resolve it.

## **Creating User Bean and User Service**

In this section, we are going to create real resources users and the post. We will use a static array list to represent the data.

**Step 1:** Create a new package with the name **com.javatpoint.server.main.user**.

**Step 2:** Create a bean class (User) to store the user detail.

Right click on the package **user -> New -> Class -> Provide Name ->** Finish. Here, we have provided class name User.

**Step 3:** Define three private variables **id, name,** and **dob**.

**Step 4:** Generate **Getters** and **Setters**.

Right click on **the file -> Source -> Generate Getters and Setters... ->Select All -> Generate**.

**Step 5:** Generate **toString**.

Advertisement

Right click on **the file -> Source -> Generate toString... ->Select All -> Generate**.

**Step 6:** Generate **Constructors**.

Right click on **the file -> Source -> Generate Constructor using Fields... -> Generate**.

**User.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.Date;
3. **public** **class** User
4. {
5. **public** User(Integer id, String name, Date dob)
6. {
7. **super**();
8. **this**.id = id;
9. **this**.name = name;
10. **this**.dob = dob;
11. }
12. **private** Integer id;
13. **private** String name;
14. **private** Date dob;
15. **public** Integer getId()
16. {
17. **return** id;
18. }
19. **public** **void** setId(Integer id)
20. {
21. **this**.id = id;
22. }
23. **public** String getName()
24. {
25. **return** name;
26. }
27. **public** **void** setName(String name)
28. {
29. **this**.name = name;
30. }
31. **public** Date getDob()
32. {
33. **return** dob;
34. }
35. **public** **void** setDob(Date dob)
36. {
37. **this**.dob = dob;
38. }
39. @Override
40. **public** String toString()
41. {
42. //return "User [id=" + id + ", name=" + name + ", dob=" + dob + "]";
43. **return** String.format("User [id=%s, name=%s, dob=%s]", id, name, dob);
44. }
45. }

Before moving to the next step first move **HelloWorldBean.java** and **HelloWorldController.java** in the package **com.javatpoint.server.main.helloworld**.

**Step 7:** Create a class with name **UserDaoService** in the package **com.javatpoint.server.main.user**.

**UserDaoService.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.ArrayList;
3. **import** java.util.Date;
4. **import** java.util.List;
5. **import** org.springframework.stereotype.Component;
6. @Component
7. **public** **class** UserDaoService
8. {
9. **public** **static** **int** usersCount=5;
10. //creating an instance of ArrayList
11. **private** **static** List<User> users=**new** ArrayList<>();
12. //static block
13. **static**
14. {
15. //adding users to the list
16. users.add(**new** User(1, "John", **new** Date()));
17. users.add(**new** User(2, "Robert", **new** Date()));
18. users.add(**new** User(3, "Adam", **new** Date()));
19. users.add(**new** User(4, "Andrew", **new** Date()));
20. users.add(**new** User(5, "Jack", **new** Date()));
21. }
22. //method that retrieve all users from the list
23. **public** List<User> findAll()
24. {
25. **return** users;
26. }
27. //method that add the user in the list
28. **public** User save(User user)
29. {
30. **if**(user.getId()==**null**)
31. {
32. //increments the user id
33. user.setId(++usersCount);
34. }
35. users.add(user);
36. **return** user;
37. }
38. //method that find a particular user from the list
39. **public** User findOne(**int** id)
40. {
41. **for**(User user:users)
42. {
43. **if**(user.getId()==id)
44. **return** user;
45. }
46. **return** **null**;
47. }
48. }

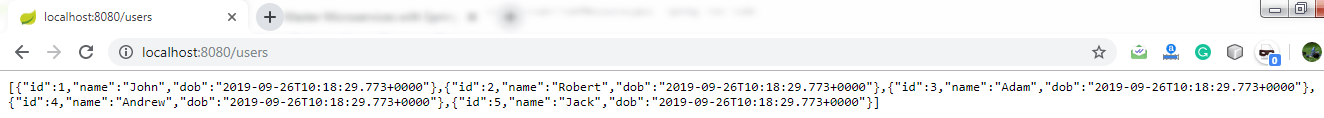
## **Implementing Get Methods for User Resource**

**Step 8:** Now create a user controller class with name **UserResource**.

**UserResource.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.GetMapping;
5. **import** org.springframework.web.bind.annotation.RestController;
6. @RestController
7. **public** **class** UserResource
8. {
9. @Autowired
10. **private** UserDaoService service;
11. @GetMapping("/users")
12. **public** List<User> retriveAllUsers()
13. {
14. **return** service.findAll();
15. }
16. }

**Step 9:** Run the application and type the **localhost:8080/users** in the address bar of the browser. It returns the users list in **JSON** format.



If the date is displaying in the default timestamp format as:

**"dob": "1500370250075"**

We need to do setting for proper date format.

Open the **application.properties** file. Remove the debug configuration and add the following configuration:

1. spring.jackson.serialization.write-dates-as-timestamps=**false**

The above statement telling the Jackson framework that when serializing don't treat the date as a timestamp.

**Step 10:** If we want to display a specific user detail on the browser, add the mapping "**/user/{id}**" and create a method **retriveUser()** in the UserResource.

**UserResource.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.GetMapping;
5. **import** org.springframework.web.bind.annotation.PathVariable;
6. **import** org.springframework.web.bind.annotation.RestController;
7. @RestController
8. **public** **class** UserResource
9. {
10. @Autowired
11. **private** UserDaoService service;
12. @GetMapping("/users")
13. **public** List<User> retriveAllUsers()
14. {
15. **return** service.findAll();
16. }
17. //retrieves a specific user detail
18. @GetMapping("/users/{id}")
19. **public** User retriveUser(@PathVariable **int** id)
20. {
21. **return** service.findOne(id);
22. }
23. }

**Step 11:** Run the application and type **localhost:8080/users/{id}** in the browser. It returns the detail of the specific user id which we are passing in the path variable.

In the following image, we have retrieved the detail of the user having id **4**.

# **Implementing the POST Method to create User Resource**

In the previous few steps, we have created simple RESTful services. In this section, we will use the POST method to post the user resource for the specific URI "**/users**."

Here we are using two annotations, **@RequestBody** and **@PathMapping**.

## **@RequestBody**

The @RequestBody annotation maps body of the web request to the method parameter. The body of the request is passed through an HttpMessageConverter. It resolves the method argument depending on the content type of the request. Optionally, automatic validation can be applied by annotating the argument with @Valid.

In the following example, when we pass the **@RequestBody** annotation in the **createUser()** method, it maps to the **user** parameter.

Backward Skip 10sPlay VideoForward Skip 10s

## **@PathMapping**

The @PathMapping annotation is the specialized version of the **@RequestMapping** annotation which acts as a shortcut for **@RequestMapping(method=RequestMethod=POST)**. @PostMapping method handles the Http POST requests matched with the specified URI.

Let's create a user resource and post that resource through the POST method.

**Step 1:** Open UserResource.java and add **@PostMapping("/user")**.

**Step 2:** Create a method **createUser()** and pass the User class's **object** as the **body** of the web.

**Step 3:** Save the created user.

**UserResource.java**

Advertisement

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.GetMapping;
5. **import** org.springframework.web.bind.annotation.PathVariable;
6. **import** org.springframework.web.bind.annotation.PostMapping;
7. **import** org.springframework.web.bind.annotation.RequestBody;
8. **import** org.springframework.web.bind.annotation.RestController;
9. @RestController
10. **public** **class** UserResource
11. {
12. @Autowired
13. **private** UserDaoService service;
14. @GetMapping("/users")
15. **public** List<user> retriveAllUsers()
16. {
17. **return** service.findAll();
18. }
19. //retrieves a specific user detail
20. @GetMapping("/users/{id}")
21. **public** User retriveUser(@PathVariable **int** id)
22. {
23. **return** service.findOne(id);
24. }
25. //method that posts a new user detail
26. @PostMapping("/users")
27. **public** **void** createUser(@RequestBody User user)
28. {
29. User sevedUser=service.save(user);
30. }
31. }
32. </user>

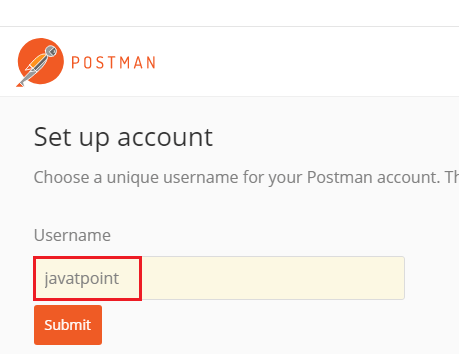
When we refresh the page it returns the get request. But we need to send POST request. We can send POST request through **REST client**. REST client is a client that is designed to use a service (RESTful) from a server.

Lets' see how to use the REST client.

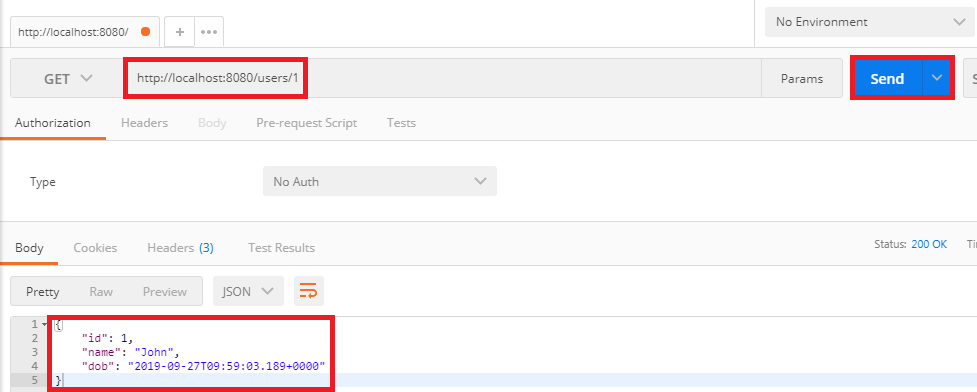
**Step 4:** Download the Postman from https://www.getpostman.com/downloads/.

Or add Google Chrome extension in the browser https://bit.ly/1HCOCwF.

Step 5: Launch the **Postman** and **Signup**. Create a **user name**. Here we have created user name **javatpoint** and clicked on **Submit** button. Consider the below image:



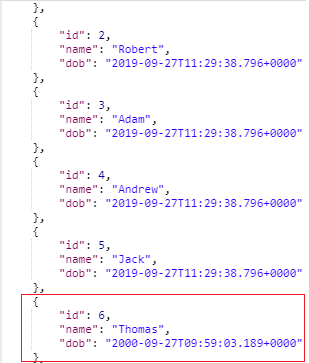
**Step 6:** First, we check for the **GET** request. Type the URL http://localhost:8080/users/1 in the address bar and click on the **Send** button. It returns the detail of the first user.



**Step 7:** Now we send a POST request.

Advertisement

* Change the method to the **POST**.
* Copy the **body** of the response coming from "/users/1".
* Click on **Body** tab. Now we create a body for the POST request.
* Choose the **raw** option. It creates a raw request.
* Paste the copied content.
* Remove the **id** because it increments automatically.
* Change the "**name**": "**Thomas**".
* Instead of Text we are sending the data in the JSON format. So select **JSON (application/json)**.
* Type the URL http://localhost:8080/users and click on the **Send** button.
* Click on the **Get** request on the left side of the window.
* Now we will send a Get request again, so change the URL http://localhost:8080/users and click on the **Send** button. It displays all the users, including which we have created.



## **Enhancing POST Method to Return Correct HTTP Code and Status Location**

In this section, we will return the status (Created) and URI ("/users/6") of the user recourse which we have created.

### **ResponseEntityClass**

The **ResponseEntity**is a class which extends **HttpEntity**and HttpStatus class. It is defined in **org.springframework.http.RequestEntity**.

* It is used in **RestTemplate** and **@Controller** methods.
* It is used as parameter in **getForEntity()** and **exchange()** method.
* It is also used in Spring MVC, as a parameter in a @Controller method.

### **created() method of RequestEntityClass**

The **created()** method is the static method of **RequestEntity**class. It creates a **new builder** with a CREATED status and a location header set to the given URI.

**Syntax**

1. **public** **static** ResponseEntity.BodyBuilder created(URI location)

**Parameter:** It accepts the **location URI** as a parameter.

**Returns:** It returns the **created builder**.

All Http status codes are **Enum constant**, which is defined in the **HttpStatus** class.

### **ServletUriComponentsBuilder**

The **ServletUriComponentsBuilder** is a class which is defined in org.springframework.web.servlet.support.ServletUriComponentsBuilder. It extends **UriComponentsBuilder** class. It has additional static factory methods to create a link based on the current HttpServletRequest.

### **fromCurrentRequest() method**

It is similar to **fromRequest(HttpServletRequest)** method except the request is obtained through RequestContextHolder.

### **path() method**

The path() is the method of **UriComponentsBuilder** class. It appends the given path to the existing path of this builder. The given path may contain URI template variable.

**Syntax**

1. **public** UriBuilderBuilder path(String path)

**Parameter:** It accepts a **path** as a parameter.

**Returns:** It returns the **UriComponentsBuilder**.

### **buildAndExpand() method**

It builds UriComponents instance and replaces URI template variables with the values obtained from an array. It is the shortcut method which combines calls to build() and then UriComponents.expand(Object... uriVariableValues).

**Syntax**

1. **public** UriComponents buildAndExpand(Object...uriVariableValues)

**Parameter:** It accepts the **URI variable values** as a parameter.

**Returns:** It returns the **URI components** with extended values.

### **build() method**

It builds UriComponents instance from the various components contained in the builder.

**Syntax**

1. **public** UriComponents build()

**Parameter:** It does not accept any parameter.

**Returns:** It returns the **Uri Components**.

Let's see how to return the status of the created resource and how to set URI of the created resource in the response.

**Step 1:** Create a method that creates a user resource and returns the **ResponseEntity**.

**UserResource.java**

1. **package** com.javatpoint.server.main.user;
2. **import** java.net.URI;
3. **import** java.util.List;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.http.ResponseEntity;
6. **import** org.springframework.web.bind.annotation.GetMapping;
7. **import** org.springframework.web.bind.annotation.PathVariable;
8. **import** org.springframework.web.bind.annotation.PostMapping;
9. **import** org.springframework.web.bind.annotation.RequestBody;
10. **import** org.springframework.web.bind.annotation.RestController;
11. **import** org.springframework.web.servlet.support.ServletUriComponentsBuilder;
12. @RestController
13. **public** **class** UserResource
14. {
15. @Autowired
16. **private** UserDaoService service;
17. @GetMapping("/users")
18. **public** List<user> retriveAllUsers()
19. {
20. **return** service.findAll();
21. }
22. //retrieves a specific user detail
23. @GetMapping("/users/{id}")
24. **public** User retriveUser(@PathVariable **int** id)
25. {
26. **return** service.findOne(id);
27. }
28. //method that posts a new user detail and returns the status of HTTP and location of the user resource
29. @PostMapping("/users")
30. **public** ResponseEntity<Object> createUser(@RequestBody User user)
31. {
32. User sevedUser=service.save(user);
33. URI location=ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(sevedUser.getId()).toUri();
34. **return** ResponseEntity.created(location).build();
35. }
36. }
37. </user>

**Step 2:** Now open REST client **Postman** and create a **POST** request.

**Step 3:** Click on the POST request under the **History** tab.

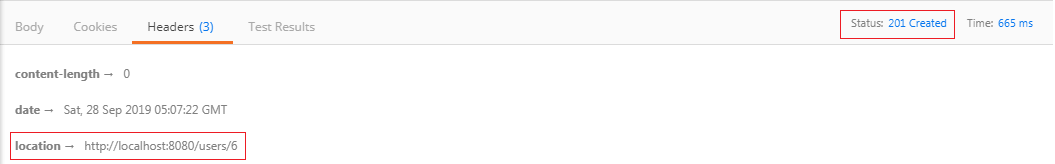
**Step 4:** Click on the Body tab and change the user name to **James**.

**Step 5:** Ensure that you have selected **JSON (application/json)** media type.

**Step 5:** Click on the **Send** button.

On the right-hand side of the window, we can see the **Status: 201 Created**. It means resource has been properly created.

**Step 6:** Now click on the Headers tab to see the location. Location is the URI of the created resource. It shows the location of the created user James that is "**/users/6**".



If the client wants to know where the user resources was created, just pick up the location from the header of the response.

# **Implementing the POST Method to create User Resource**

In the previous few steps, we have created simple RESTful services. In this section, we will use the POST method to post the user resource for the specific URI "**/users**."

Here we are using two annotations, **@RequestBody** and **@PathMapping**.

## **@RequestBody**

The @RequestBody annotation maps body of the web request to the method parameter. The body of the request is passed through an HttpMessageConverter. It resolves the method argument depending on the content type of the request. Optionally, automatic validation can be applied by annotating the argument with @Valid.

In the following example, when we pass the **@RequestBody** annotation in the **createUser()** method, it maps to the **user** parameter.

Backward Skip 10sPlay VideoForward Skip 10s

## **@PostMapping**

The @PostMapping annotation is the specialized version of the **@RequestMapping** annotation which acts as a shortcut for **@RequestMapping(method=RequestMethod=POST)**. @PostMapping method handles the Http POST requests matched with the specified URI.

Let's create a user resource and post that resource through the POST method.

**Step 1:** Open UserResource.java and add **@PostMapping("/user")**.

**Step 2:** Create a method **createUser()** and pass the User class's **object** as the **body** of the web.

**Step 3:** Save the created user.

**UserResource.java**

Advertisement

1. **package** com.javatpoint.server.main.user;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.GetMapping;
5. **import** org.springframework.web.bind.annotation.PathVariable;
6. **import** org.springframework.web.bind.annotation.PostMapping;
7. **import** org.springframework.web.bind.annotation.RequestBody;
8. **import** org.springframework.web.bind.annotation.RestController;
9. @RestController
10. **public** **class** UserResource
11. {
12. @Autowired
13. **private** UserDaoService service;
14. @GetMapping("/users")
15. **public** List<user> retriveAllUsers()
16. {
17. **return** service.findAll();
18. }
19. //retrieves a specific user detail
20. @GetMapping("/users/{id}")
21. **public** User retriveUser(@PathVariable **int** id)
22. {
23. **return** service.findOne(id);
24. }
25. //method that posts a new user detail
26. @PostMapping("/users")
27. **public** **void** createUser(@RequestBody User user)
28. {
29. User sevedUser=service.save(user);
30. }
31. }
32. </user>

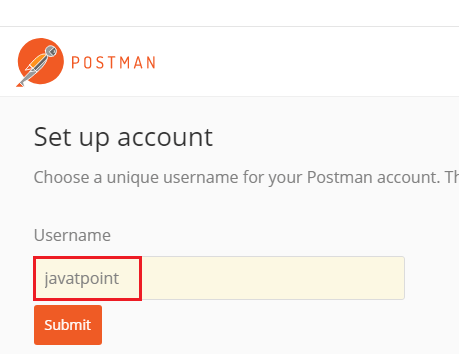
When we refresh the page it returns the get request. But we need to send POST request. We can send POST request through **REST client**. REST client is a client that is designed to use a service (RESTful) from a server.

Lets' see how to use the REST client.

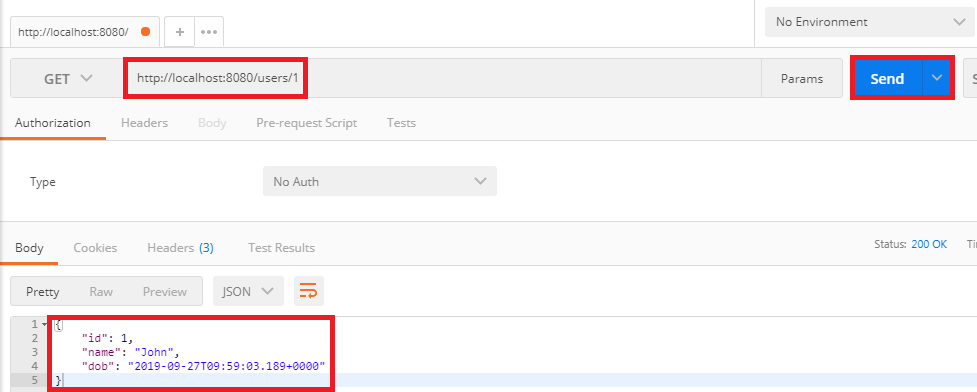
**Step 4:** Download the Postman from https://www.getpostman.com/downloads/.

Or add Google Chrome extension in the browser https://bit.ly/1HCOCwF.

Step 5: Launch the **Postman** and **Signup**. Create a **user name**. Here we have created user name **javatpoint** and clicked on **Submit** button. Consider the below image:



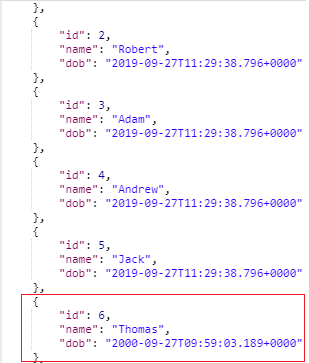
**Step 6:** First, we check for the **GET** request. Type the URL http://localhost:8080/users/1 in the address bar and click on the **Send** button. It returns the detail of the first user.



**Step 7:** Now we send a POST request.

Advertisement

* Change the method to the **POST**.
* Copy the **body** of the response coming from "/users/1".
* Click on **Body** tab. Now we create a body for the POST request.
* Choose the **raw** option. It creates a raw request.
* Paste the copied content.
* Remove the **id** because it increments automatically.
* Change the "**name**": "**Thomas**".
* Instead of Text we are sending the data in the JSON format. So select **JSON (application/json)**.
* Type the URL http://localhost:8080/users and click on the **Send** button.
* Click on the **Get** request on the left side of the window.
* Now we will send a Get request again, so change the URL http://localhost:8080/users and click on the **Send** button. It displays all the users, including which we have created.



## **Enhancing POST Method to Return Correct HTTP Code and Status Location**

In this section, we will return the status (Created) and URI ("/users/6") of the user recourse which we have created.

### **ResponseEntityClass**

The **ResponseEntity**is a class which extends **HttpEntity**and HttpStatus class. It is defined in **org.springframework.http.RequestEntity**.

* It is used in **RestTemplate** and **@Controller** methods.
* It is used as parameter in **getForEntity()** and **exchange()** method.
* It is also used in Spring MVC, as a parameter in a @Controller method.

### **created() method of RequestEntityClass**

The **created()** method is the static method of **RequestEntity**class. It creates a **new builder** with a CREATED status and a location header set to the given URI.

**Syntax**

1. **public** **static** ResponseEntity.BodyBuilder created(URI location)

**Parameter:** It accepts the **location URI** as a parameter.

Advertisement

**Returns:** It returns the **created builder**.

All Http status codes are **Enum constant**, which is defined in the **HttpStatus** class.

### **ServletUriComponentsBuilder**

The **ServletUriComponentsBuilder** is a class which is defined in org.springframework.web.servlet.support.ServletUriComponentsBuilder. It extends **UriComponentsBuilder** class. It has additional static factory methods to create a link based on the current HttpServletRequest.

### **fromCurrentRequest() method**

It is similar to **fromRequest(HttpServletRequest)** method except the request is obtained through RequestContextHolder.

### **path() method**

The path() is the method of **UriComponentsBuilder** class. It appends the given path to the existing path of this builder. The given path may contain URI template variable.

**Syntax**

1. **public** UriBuilderBuilder path(String path)

Advertisement

**Parameter:** It accepts a **path** as a parameter.

**Returns:** It returns the **UriComponentsBuilder**.

### **buildAndExpand() method**

It builds UriComponents instance and replaces URI template variables with the values obtained from an array. It is the shortcut method which combines calls to build() and then UriComponents.expand(Object... uriVariableValues).

**Syntax**

1. **public** UriComponents buildAndExpand(Object...uriVariableValues)

**Parameter:** It accepts the **URI variable values** as a parameter.

**Returns:** It returns the **URI components** with extended values.

### **build() method**

It builds UriComponents instance from the various components contained in the builder.

**Syntax**

1. **public** UriComponents build()

**Parameter:** It does not accept any parameter.

**Returns:** It returns the **Uri Components**.

Let's see how to return the status of the created resource and how to set URI of the created resource in the response.

**Step 1:** Create a method that creates a user resource and returns the **ResponseEntity**.

**UserResource.java**

Advertisement

1. **package** com.javatpoint.server.main.user;
2. **import** java.net.URI;
3. **import** java.util.List;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.http.ResponseEntity;
6. **import** org.springframework.web.bind.annotation.GetMapping;
7. **import** org.springframework.web.bind.annotation.PathVariable;
8. **import** org.springframework.web.bind.annotation.PostMapping;
9. **import** org.springframework.web.bind.annotation.RequestBody;
10. **import** org.springframework.web.bind.annotation.RestController;
11. **import** org.springframework.web.servlet.support.ServletUriComponentsBuilder;
12. @RestController
13. **public** **class** UserResource
14. {
15. @Autowired
16. **private** UserDaoService service;
17. @GetMapping("/users")
18. **public** List<user> retriveAllUsers()
19. {
20. **return** service.findAll();
21. }
22. //retrieves a specific user detail
23. @GetMapping("/users/{id}")
24. **public** User retriveUser(@PathVariable **int** id)
25. {
26. **return** service.findOne(id);
27. }
28. //method that posts a new user detail and returns the status of HTTP and location of the user resource
29. @PostMapping("/users")
30. **public** ResponseEntity<Object> createUser(@RequestBody User user)
31. {
32. User sevedUser=service.save(user);
33. URI location=ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(sevedUser.getId()).toUri();
34. **return** ResponseEntity.created(location).build();
35. }
36. }
37. </user>

**Step 2:** Now open REST client **Postman** and create a **POST** request.

**Step 3:** Click on the POST request under the **History** tab.

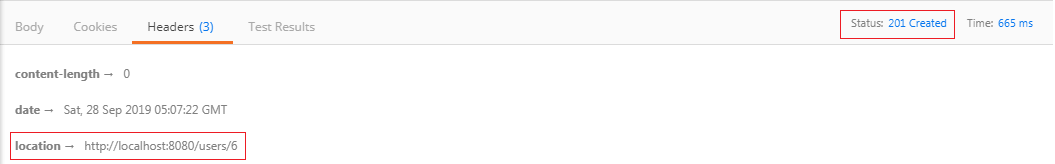
**Step 4:** Click on the Body tab and change the user name to **James**.

**Step 5:** Ensure that you have selected **JSON (application/json)** media type.

**Step 5:** Click on the **Send** button.

On the right-hand side of the window, we can see the **Status: 201 Created**. It means resource has been properly created.

**Step 6:** Now click on the Headers tab to see the location. Location is the URI of the created resource. It shows the location of the created user James that is "**/users/6**".



If the client wants to know where the user resources was created, just pick up the location from the header of the response.

# **Implementing Exception Handling- 404 Resource Not Found**

In the previous section, we had returned the proper response status of CREATED when we created the resource. In this section, we will discuss what should be the response when a user resource does not exist.

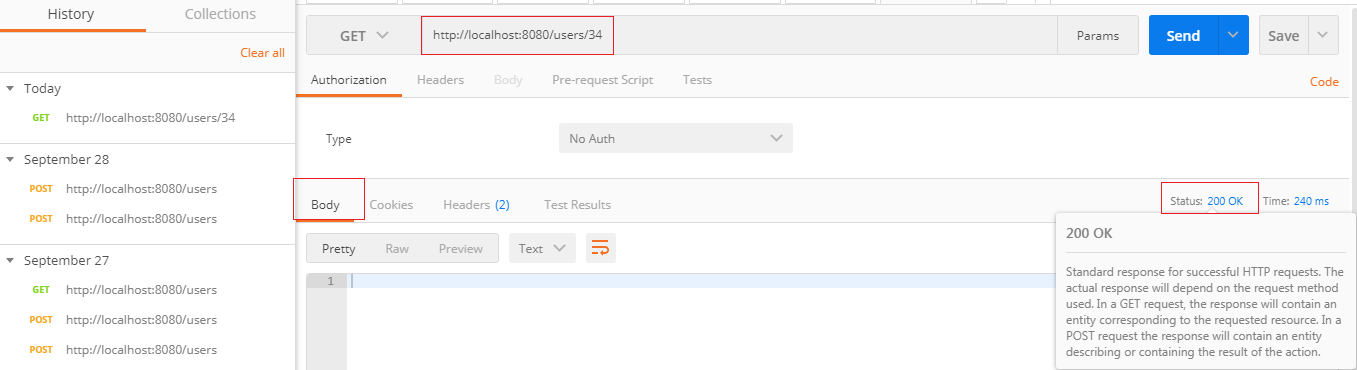
Let's try and execute a simple response.

**Step 1:** Open Rest client **Postman** and select the **Get** method.

**Step 2:** Click on the **History** tab and choose the **Get** request.

**Step 3:** Type the URI http://localhost:8080/users/{id}. The user id should not exist.

**Step 4:** Click on the **Send** Button.



We get the **Status: 200 OK** and **empty body** which is a successful response even though the resource does not exist. But it is not the proper response when a resource does not exist.

Let's fix that first.

**Step 1:** Open the **UserResource.java** file.

**Step 2:** Create a **UserNotFoundException**. It is a checked exception.

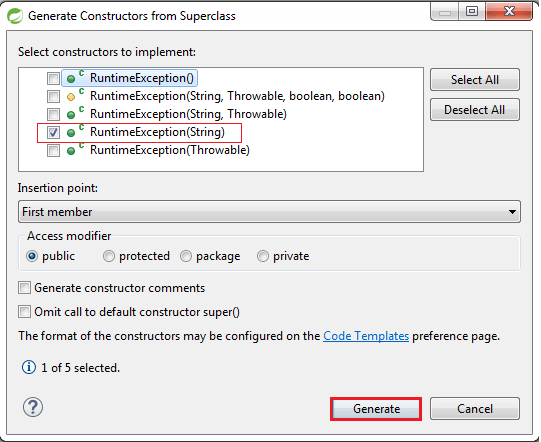
Advertisement

1. @GetMapping("/users/{id}")
2. **public** User retriveUser(@PathVariable **int** id)
3. {
4. User user= service.findOne(id);
5. **if**(user==**null**)
6. //runtime exception
7. **throw** **new** UserNotFoundException("id: "+ id);
8. **return** user;
9. }

**Step 3:** Create **UserNotFoundException** class.

**Step 4:** Generate Constructors from Superclass.

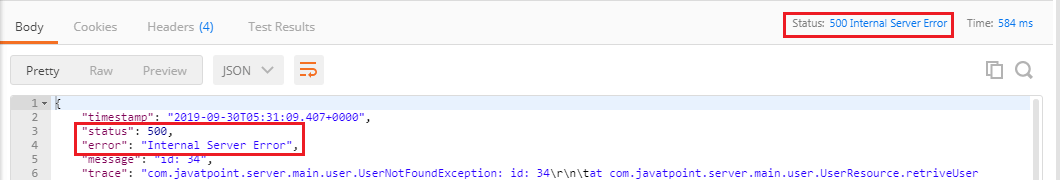
Right-click on the file -> Source -> Generate Constructors from Superclass... -> check the RuntimeException(String) -> Generate.



**UserNotFoundException.java**

1. **package** com.javatpoint.server.main.user;
2. **public** **class** UserNotFoundException **extends** RuntimeException
3. {
4. **public** UserNotFoundException(String message)
5. {
6. **super**(message);
7. }
8. }

**Step 5:** Open the Rest Client **Postman** and generate a Get response as we have done before. It shows the **Status: 500 Internal Server Error**.

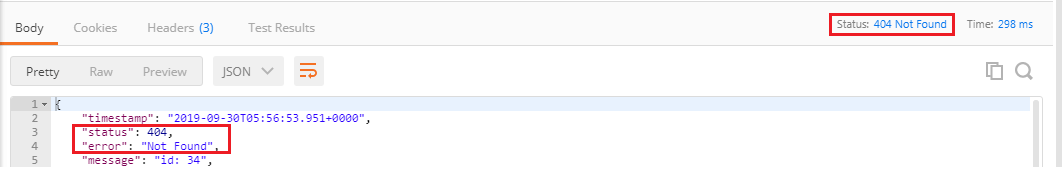


But the Status: 500 Internal Server Error is not the appropriate response for the resource not found. So, we will add an annotation **@ResponseStatus** to generate the Status: 404 Not Found.

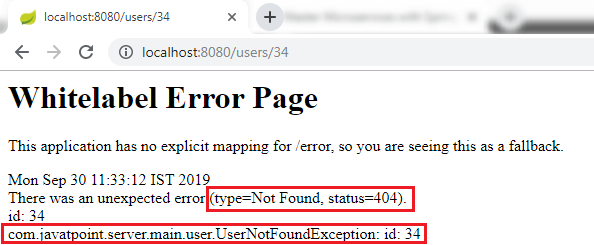
**UserNotFoundException.java**

1. **package** com.javatpoint.server.main.user;
2. **import** org.springframework.http.HttpStatus;
3. **import** org.springframework.web.bind.annotation.ResponseStatus;
4. @ResponseStatus(HttpStatus.NOT\_FOUND)
5. **public** **class** UserNotFoundException **extends** RuntimeException
6. {
7. **public** UserNotFoundException(String message)
8. {
9. **super**(message);
10. }
11. }

**Step 6:** Again move to **Postman** and generate a **Get** request.



We get the proper response **Status: 404 Not Found** when a user resource does not exist. The body of the request provided by default error handling that's why we are getting this return status back.



The combination of Spring Boot and Spring Web MVC framework provides error handling. Spring Boot auto-configures some default exception handling. It is important to have a consistent exception message which is obtained for all the services inside our enterprise.

If we have a big organization and each of the services returns the exception messages in a different way, so it is not good. It would be good if we define a standard exception structure which is followed by across all the RESTful Services.

[Click here to download Not Found project](https://static.javatpoint.com/tutorial/restful-web-services/download/Not%20Found.zip)

[Click here to download 500 Internal Server Error project](https://static.javatpoint.com/tutorial/restful-web-services/download/500%20Internal%20Server%20Error.zip)

# **Implementing Generic Exception Handling for all Resources**

As we discussed earlier that we should define a standard exception structure which is followed across all the RESTful services. In this section, we will discuss the implementation **of generic exception handling for all resources**.

Let's see how to customize the exception message.

**Step 1:** Create a new package with name **com.javatpoint.server.main.exception**.

**Step 2:** Create a class with the name **ExceptionResponse** in the above package.

**Step 3:** At the basic level, there are three crucial things for the exception structure: **timestamp, message,** and **detail**. Define these three fields.

**Step 4:** Generate Constructors using Fields.

**Step 5:** Generate Getters, Setters are not needed.

# Once the structure is defined, we can define Java implementation for it. Whenever an exception occurs, we would return a response in the specific format. The structure is the most important part and must have to be language independent.

One of the important class in Spring Framework is **ResponseEntityExceptionHandler** class. It is an abstract and base class for the exceptions that provide centralized exception handling across all the different exception handler methods. We will extend this class to handle and provide customized exception handling functionality. This exception handling functionality is applied to all controller as HellowWorldController, UserResource (Controller).

**Step 6:** Create a new class with name **CustomizedResponseEntityExceptionHandler** in the package **com.javatpoint.server.main.exception** and extends **ResponseEntityExceptionHandler** class.

Advertisement

**Step 7:** Add @ControllerAdvice and @RestController annotations.

**Step 8:** Expand Maven Dependencies in the package explorer -> Expand spring-webmvc-5.1.9.RELEASE.jar -> Expand org.springframework.web.servlet.mvc.method.annotation -> Open **ResponseEntityExceptionHandler.class**.

Implementing Generic Exception Handling for all Resources

**Step 9:** Copy the **ResponseEntityMethod<Object>** method from ResponseEntityExceptionHandler.class and paste in **CustomizedResponseEntityExceptionHandler.java** file.

**Step 10:** Override the **ResponseEntityMethod** method. Rename the method name as handleAllExceptions().

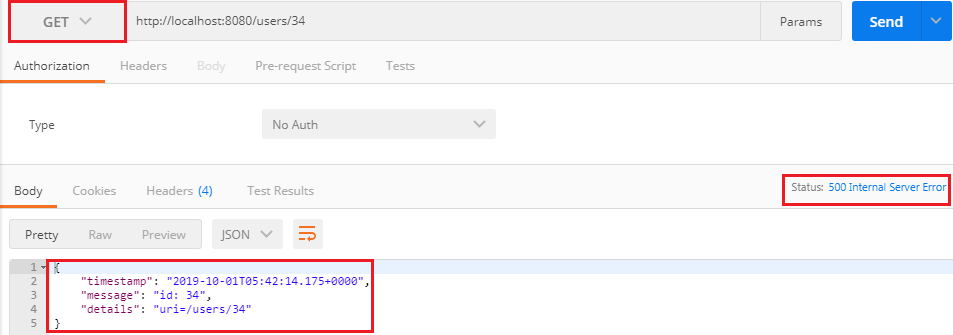
**Step 11:** Create the exception response structure.

**Step 12:** Create a **ResponseEntity** object and pass the exception response and Http status as arguments.

**CustomizedResponseEntityExceptionHandler.java**

1. **package** com.javatpoint.server.main;
2. **import** java.util.Date;
3. **import** org.springframework.http.HttpStatus;
4. **import** org.springframework.http.ResponseEntity;
5. **import** org.springframework.web.bind.annotation.ControllerAdvice;
6. **import** org.springframework.web.bind.annotation.ExceptionHandler;
7. **import** org.springframework.web.bind.annotation.RestController;
8. **import** org.springframework.web.context.request.WebRequest;
9. **import** org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;
10. **import** com.javatpoint.server.main.exception.ExceptionResponse;
11. //defining exception handling for all the exceptions
12. @ControllerAdvice
13. @RestController
14. **public** **class** CustomizedResponseEntityExceptionHandler **extends** ResponseEntityExceptionHandler
15. {
16. @ExceptionHandler(Exception.**class**)
17. //override method of ResponseEntityExceptionHandler class
18. **public** **final** ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest request)
19. {
20. //creating exception response structure
21. ExceptionResponse exceptionResponse= **new** ExceptionResponse(**new** Date(), ex.getMessage(), request.getDescription(**false**));
22. //returning exception structure and specific status
23. **return** **new** ResponseEntity(exceptionResponse, HttpStatus.INTERNAL\_SERVER\_ERROR);
24. }
25. }

**Step 13:** Open rest client **Postman** and send a **Get** request. We get the **Status: 500 Internal Server Error** and exception structure which we have defined.



If we want to customize the status **Internal Server Error to Not Found**, we are required to change a few things in **CustomizedResponseEntityExceptionHandler.java** file.

**CustomizedResponseEntityExceptionHandler.java**

1. **package** com.javatpoint.server.main;
2. **import** java.util.Date;
3. **import** org.springframework.http.HttpStatus;
4. **import** org.springframework.http.ResponseEntity;
5. **import** org.springframework.web.bind.annotation.ControllerAdvice;
6. **import** org.springframework.web.bind.annotation.ExceptionHandler;
7. **import** org.springframework.web.bind.annotation.RestController;
8. **import** org.springframework.web.context.request.WebRequest;
9. **import** org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;
10. **import** com.javatpoint.server.main.exception.ExceptionResponse;
11. **import** com.javatpoint.server.main.user.UserNotFoundException;
12. //defining exception handling for all the exceptions
13. @ControllerAdvice
14. @RestController
15. **public** **class** CustomizedResponseEntityExceptionHandler **extends** ResponseEntityExceptionHandler
16. {
17. @ExceptionHandler(Exception.**class**)
18. //override method of ResponseEntityExceptionHandler class
19. **public** **final** ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest request)
20. {
21. //creating exception response structure
22. ExceptionResponse exceptionResponse= **new** ExceptionResponse(**new** Date(), ex.getMessage(), request.getDescription(**false**));
23. //returning exception structure and Internal Server status
24. **return** **new** ResponseEntity(exceptionResponse, HttpStatus.INTERNAL\_SERVER\_ERROR);
25. }
26. @ExceptionHandler(UserNotFoundException.**class**)
27. //override method of ResponseEntityExceptionHandler class
28. **public** **final** ResponseEntity<Object> handleUserNotFoundExceptions(UserNotFoundException ex, WebRequest request)
29. {
30. //creating exception response structure
31. ExceptionResponse exceptionResponse= **new** ExceptionResponse(**new** Date(), ex.getMessage(), request.getDescription(**false**));
32. //returning exception structure and Not Found status
33. **return** **new** ResponseEntity(exceptionResponse, HttpStatus.NOT\_FOUND);
34. }
35. }

Again move to **Postman** and send a **Get** request. We get the **Status: 404 Not Found** with the defined exception structure.

