COURSEWARE

Controllers

Contents

Overview

Controllers allow for external access to a Spring application via HTTP requests.

@RestController

Allows for the use of `RequestMapping` to expose methods and provides configuration for handling requests in a RESTful manner (namely returning data in a JSON format).

```
@RestController
public class UserController {

}
```

@RequestMapping

Exposes methods to requests at a defined URL, for example:

```
    @RequestMapping("/getAll")
    public List<User> getAllUsers() {
        //
    }
```

This code would map *any* received request at **/getAll** to the `getAllUsers` method. In a `RestController`, returning `List<User>` will generate a JSON array of `User` objects in the body of the HTTP response.

When the `@RequestMapping` annotation is used without a specific type of request such as `GET` or `POST`, then the method will work for all types of request.
Most of the time we will want to map methods to a specific *type* of request. Spring provides several annotations that allow us to do this.

@GetMapping

Maps methods to **GET** requests, which are used to *fetch* data.

Eg. `GET: http://localhost:8080/user/getAll`

```
@GetMapping("/getAll")
public List<User> getAllUsers() {
    // ...
}
```

@PutMapping

Maps methods to **PUT** requests, which are used to *replace* data in its entirety.

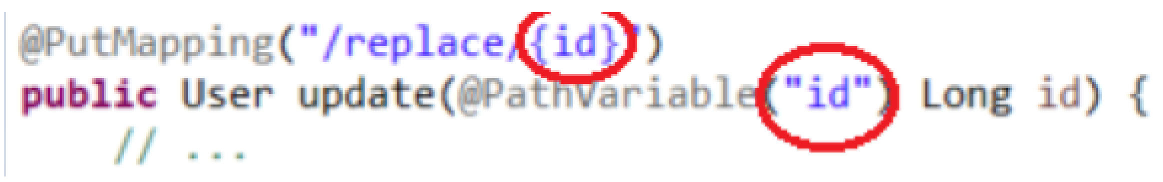
Eg. `PUT: http://localhost:8080/user/replace/1?firstName=barry&lastName=scott`

```
@PutMapping("/replace/{id}")
public User replace(@PathParam("firstName") String firstName,
@PathParam("lastName") String lastName, @PathVariable Long id) {
    // ...
}
```

@PathVariable

Extracts values from the URL the request was sent to.
If the value in `{}` matches the parameter name then the value will be inserted automatically, otherwise, the parameter can be specified.

The usage of `PathVariable` in the last example is equivalent to using:



```
@PutMapping("/replace/{id}")
public User update(@PathVariable("id") Long id) {
    // ...
```

@PathParam

`@PathParam` is a parameter annotation which allows you to map query parameters in the request to parameters in the method.

```
@GetMapping("/user/{id}")
public User getUserById(@PathParm("id") String id) {
    // ...
}
```

@PostMapping

Maps methods to **POST** requests, which are used to *send* data.

Eg: `POST: http://localhost:8080/user/register`

The data that is being sent is in JSON Format, the body will look something similar to this:

```
{
    "firstName": "Tim",
    "lastName": "Taylor",
    "userName": "timT76",
    "password": "jmgmghkmky54646"
}
```

```
@PostMapping("/register")
public User register(@RequestBody User user) {
    // ...
}
```

Given that Spring maps the data that is received via the body to the `User` class, the field names of the class need to match the JSON format otherwise there will be a failure in mapping.

▶ User.java

@RequestBody

Data sent in the body of a request can be converted from JSON to a java object by marking a method parameter with `@RequestBody`.
Only *one* parameter can be marked this way.

@PatchMapping

Maps methods to PATCH requests, which are used to **partially** *update* data.

Eg. `PATCH: http://localhost:8080/user/replace/1?username=bscott&password=pass123`

```
@PatchMapping("/update/{id}")
public User replace(@PathParam("username") String username,
@PathParam("password"), @PathVariable Long id) {
    // ...
}
```

A `PATCH` request differs from a `PUT` request as it targets specific information to change, rather than replacing the whole resource.

@DeleteMapping

Maps methods to DELETE requests, which are used to *remove* data.

Eg. `DELETE: http://localhost:8080/delete/1`

```
@DeleteMapping("/delete/{id}")
public boolean delete(@PathVariable Long id) {
    // ...
}
```

ResponseEntity

Allows for the creation of HTTP responses using Java objects.
Using `ResponseEntity` the headers, body and status code of the response can be configured.

In the header configuration, the user might want to set specific values for their responses.
Some examples include:

- Status code.
- Content-type.
- Access Control.

```

@GetMapping("/get/{id}")
public ResponseEntity<User> getUser(@PathVariable Long id) {
    return new ResponseEntity<User>(this.service.getUser(id), HTTPStatus.OK);
}
```

Tutorial

For this tutorial we will be creating a simple **Controller** that modifies a `List` rather than interacting with a database.

Start with creating a new **Spring Starter Project** - make sure you have the **Spring Web** dependency!

Create a `PersonController` in an appropriate package.

```
public class PersonController {

}
```

First thing to do with any Spring Component is to add the correct annotation, in this case `@RestController`

```
@RestController
public class PersonController {

}
```

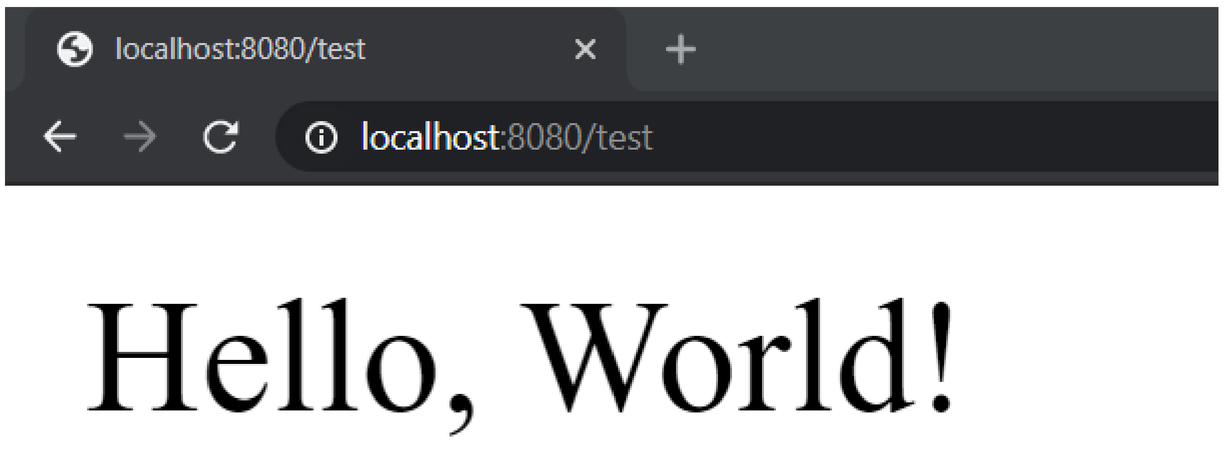
Next thing is to create a test endpoint to check what we've done so far

```
@RestController
public class PersonController {

    @GetMapping("/test")
    public String test() {
        return "Hello, World!";
    }

}
```

Now start the application then go to `http://localhost:8080/test` either in Postman or your web browser.



Hello, World!

Once we know the Controller works we can start implementing our basic functionality, starting with the `List`.

```
@RestController
public class PersonController {

    private List<Person> people = new ArrayList<>();

    @GetMapping("/test")
    public String test() {
        return "Hello, World!";
    }

}
```

Now we can create the necessary methods and mappings to implement CRUD functionality in the `PersonController` using the `people List`.

Create

1. Make the `@PostMapping`

```
    @PostMapping("/create")
```

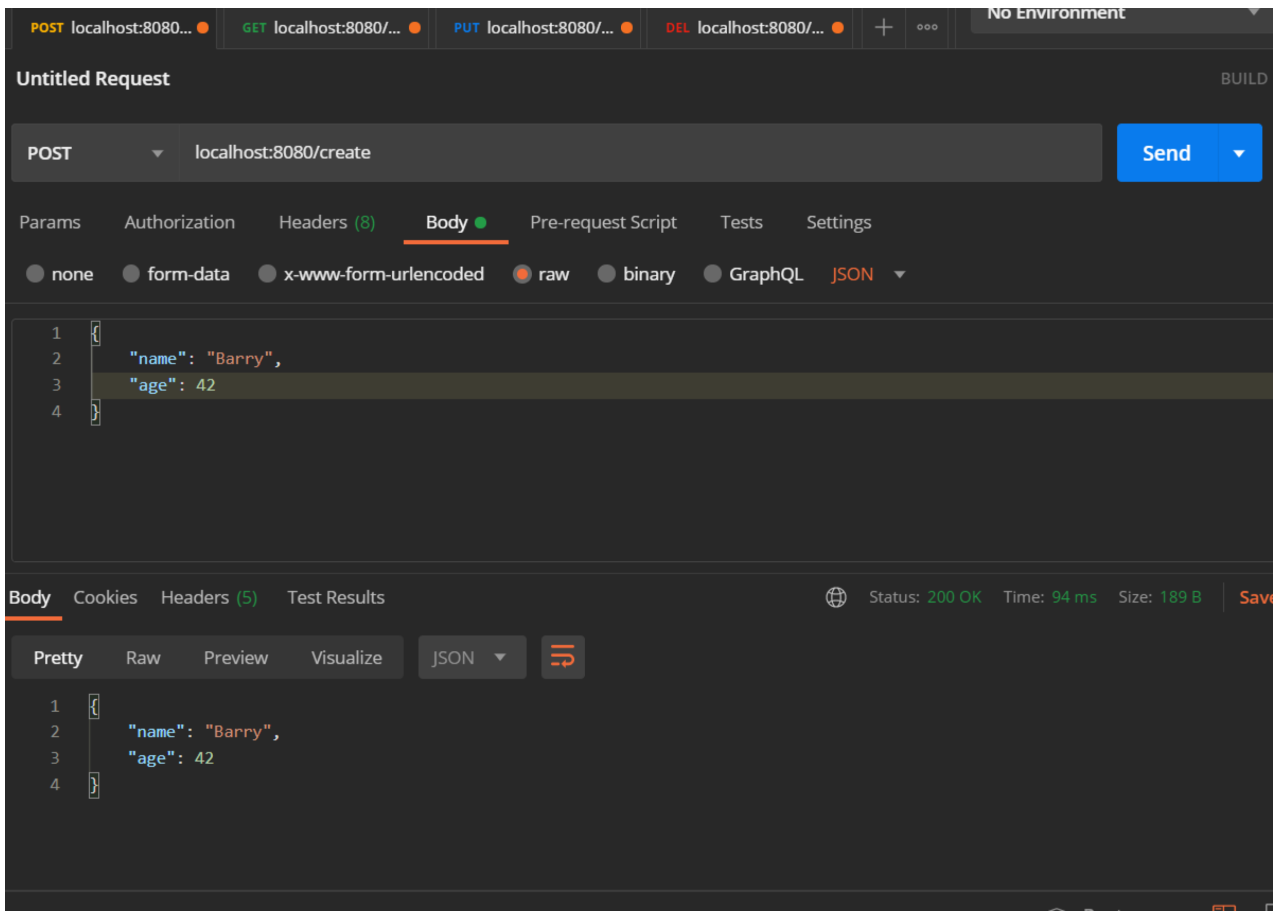
2. Create the basic method

```
    // Create
    @PostMapping("/create")
    public boolean addPerson(Person person) {
        return this.people.add(person);
    }
```

3. Add the `@RequestBody`

```
    // Create
    @PostMapping("/create")
    public boolean addPerson(@RequestBody Person person) {
        return this.people.add(person);
    }
```

4. Test in Postman


POST localhost:8080... GET localhost:8080/... PUT localhost:8080/... DEL localhost:8080/... No Environment

Untitled Request BUILD

POST localhost:8080/create Send

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{
    "name": "Barry",
    "age": 42
}
```

Body Cookies Headers (5) Test Results Status: 200 OK Time: 94 ms Size: 189 B Save

Pretty Raw Preview Visualize JSON

```
{
    "name": "Barry",
    "age": 42
}
```

Read

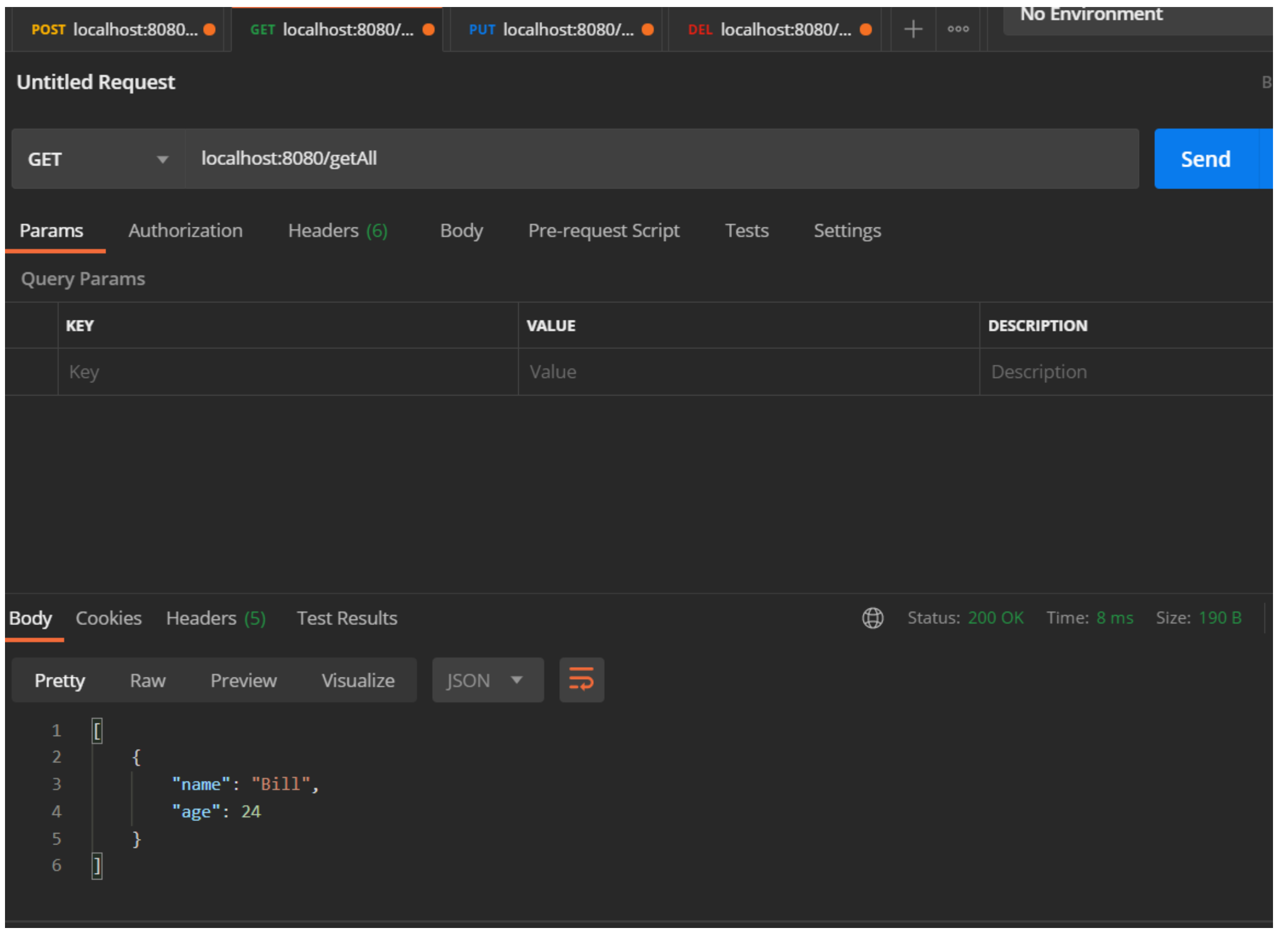
1. Make the `@GetMapping`

```
@GetMapping("/getAll")
```

2. Create the basic method

```
    // READ
    @GetMapping("/getAll")
    public List<Person> getAll() {
        return this.people;
    }
```

3. Test in Postman


POST localhost:8080... GET localhost:8080/... PUT localhost:8080/... DEL localhost:8080/... No Environment

Untitled Request

GET localhost:8080/getAll Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (5) Test Results Status: 200 OK Time: 8 ms Size: 190 B

Pretty Raw Preview Visualize JSON

```
[
    {
        "name": "Bill",
        "age": 24
    }
]
```

Update

1. Add the `@PutMapping` - we're using **PUT** as this will be a *replacement* not just a partial update.

```
@PutMapping("/update")
```

2. Create the basic method

```
    @PutMapping("/update")
    public Person updatePerson(int id, Person person) {
        // Remove existing Person with matching 'id'
        this.people.remove(id);
        // Add new Person in its place
        this.people.add(id, person);
        // Return updated Person from List
        return this.people.get(id);
    }
```

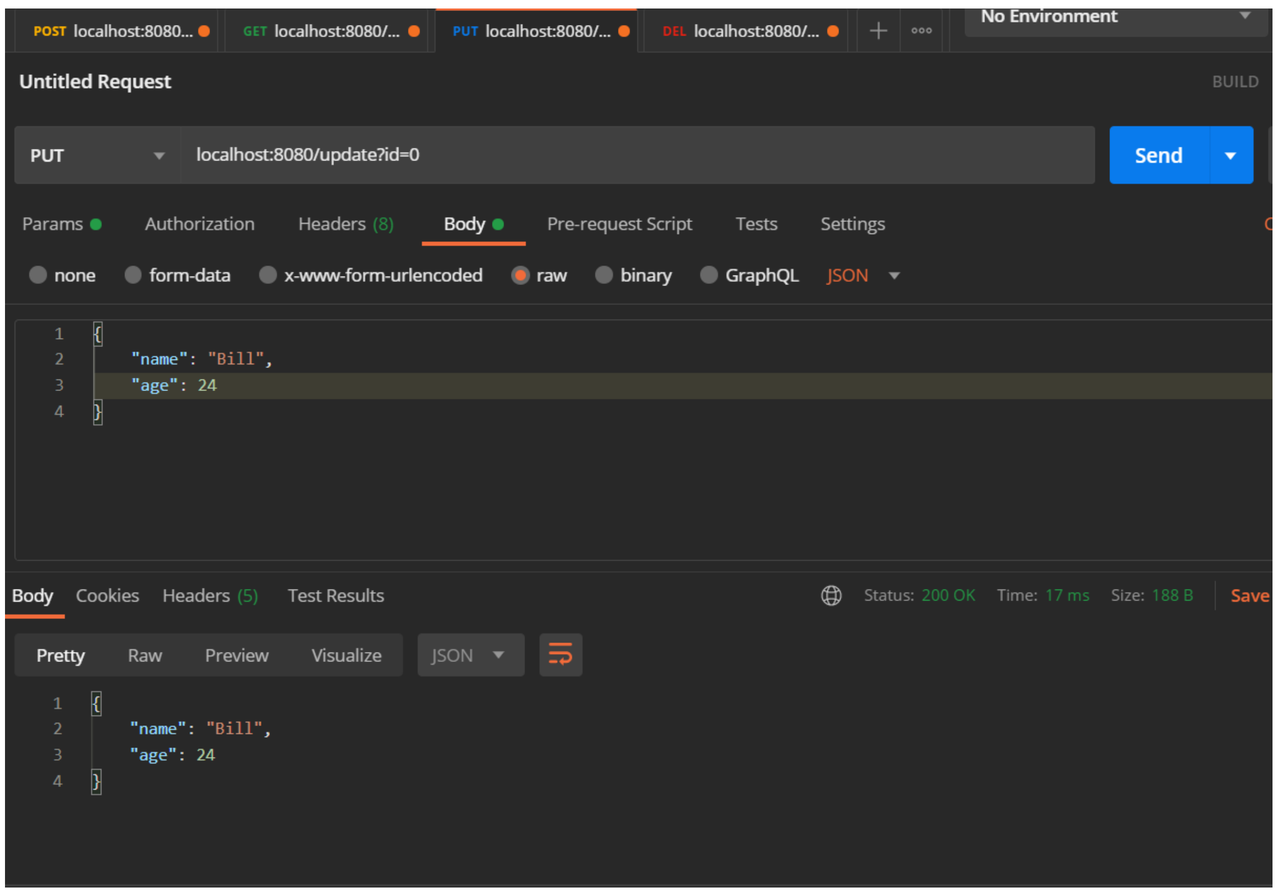
3. Add the `@PathParam`

```
    @PutMapping("/update")
    public Person updatePerson(@PathParam("id") int id, Person person) {
        // Remove existing Person with matching 'id'
        this.people.remove(id);
        // Add new Person in its place
        this.people.add(id, person);
        // Return updated Person from List
        return this.people.get(id);
    }
```

4. Add the `@RequestBody`

```
    @PutMapping("/update")
    public Person updatePerson(@PathParam("id") int id, @RequestBody Person
person) {
        // Remove existing Person with matching 'id'
        this.people.remove(id);
        // Add new Person in its place
        this.people.add(id, person);
        // Return updated Person from List
        return this.people.get(id);
    }
```

5. Test in Postman



PUT localhost:8080/update?id=0

```
{
    "name": "Bill",
    "age": 24
}
```

Status: 200 OK Time: 17 ms Size: 188 B

```
{
    "name": "Bill",
    "age": 24
}
```

Delete

1. Add the `@DeleteMapping`

```
@DeleteMapping("/delete/{id}")
```

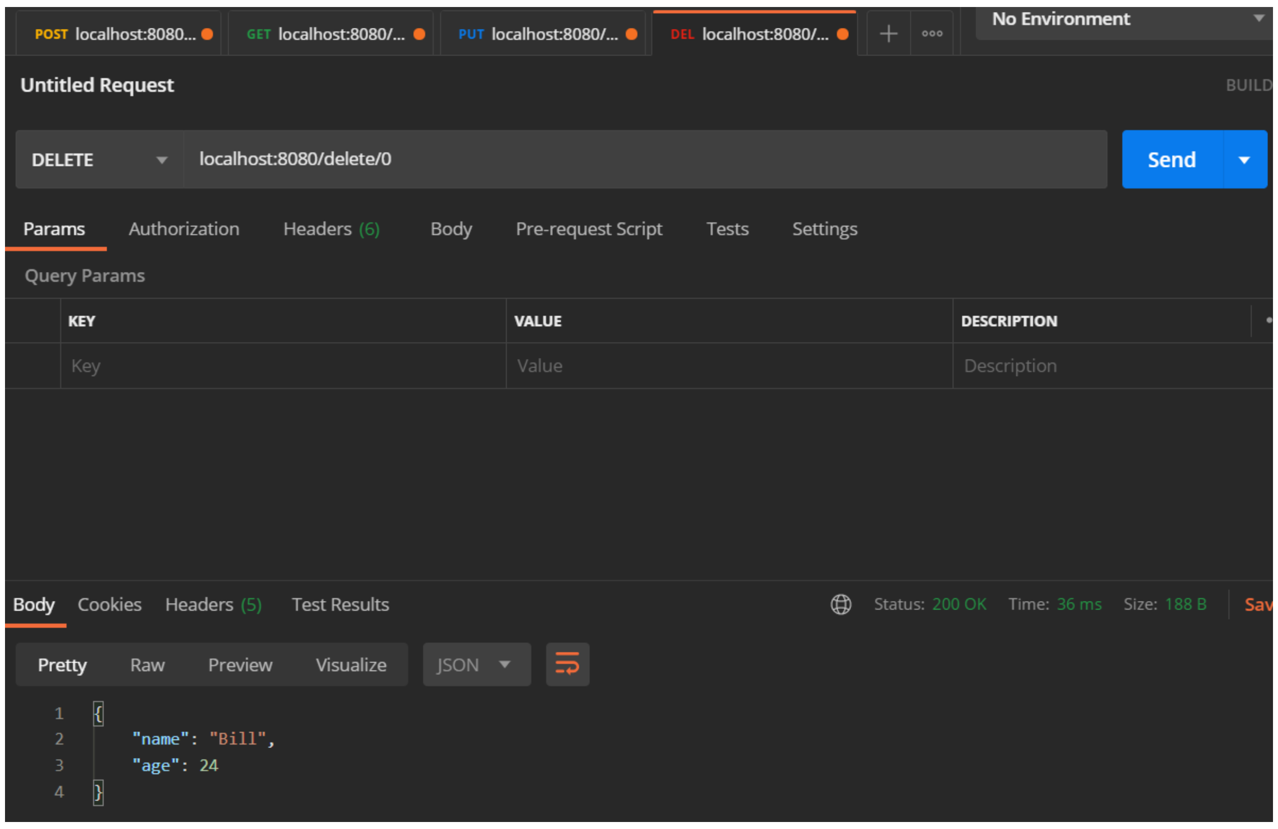
2. Create the basic method

```
@DeleteMapping("/delete/{id}")
public Person removePerson(int id) {
    // Remove Person and return it
    return this.people.remove(id);
}
```

3. Add the `@PathVariable`

```
@DeleteMapping("/delete/{id}")
public Person removePerson(@PathVariable int id) {
    // Remove Person and return it
    return this.people.remove(id);
}
```

4. Test in Postman



POST localhost:8080... GET localhost:8080/... PUT localhost:8080/... DEL localhost:8080/...

No Environment

Untitled Request

DELETE localhost:8080/delete/0 Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (5) Test Results Status: 200 OK Time: 36 ms Size: 188 B

Pretty Raw Preview Visualize JSON

```
{
    "name": "Bill",
    "age": 24
}
```

Exercises

Using the project you created in the Entities module create an `AccountController` amd implement full CRUD functionality using a `List`.