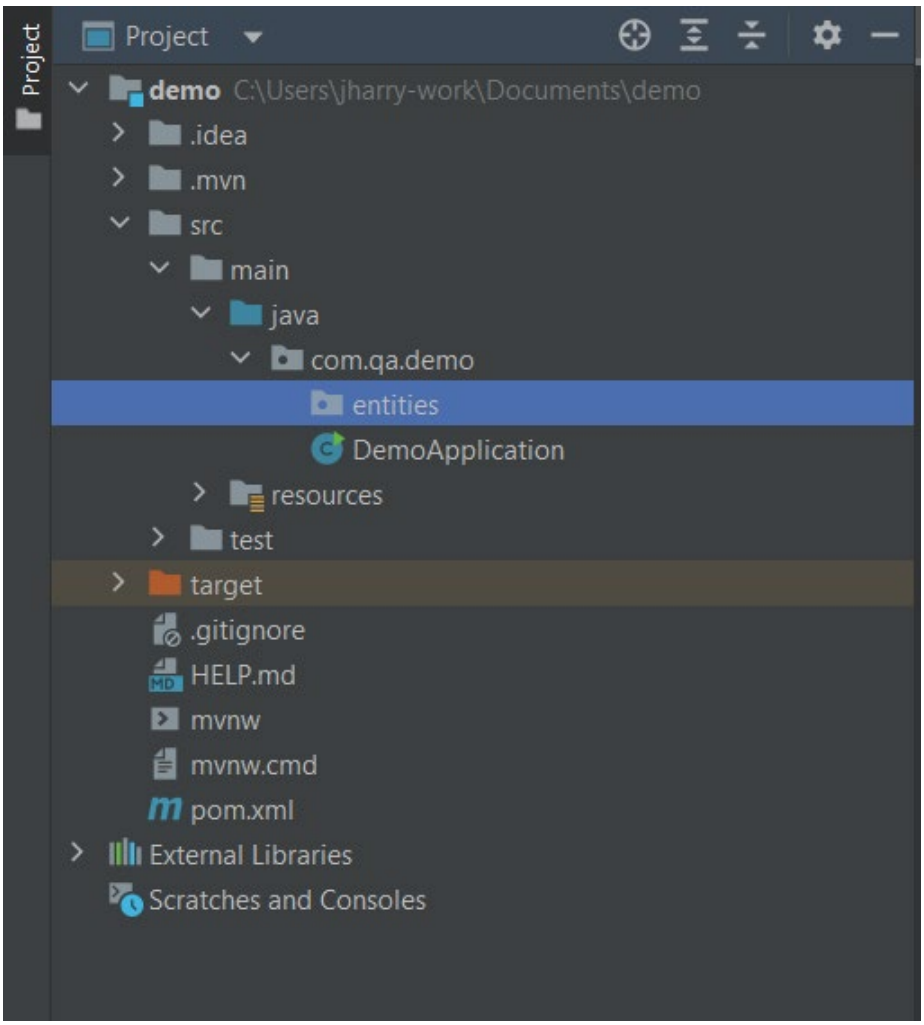
Lab 2 – Setting up the controller

The purpose of this task is to setup a basic RESTful API using Spring Web.

Task 1

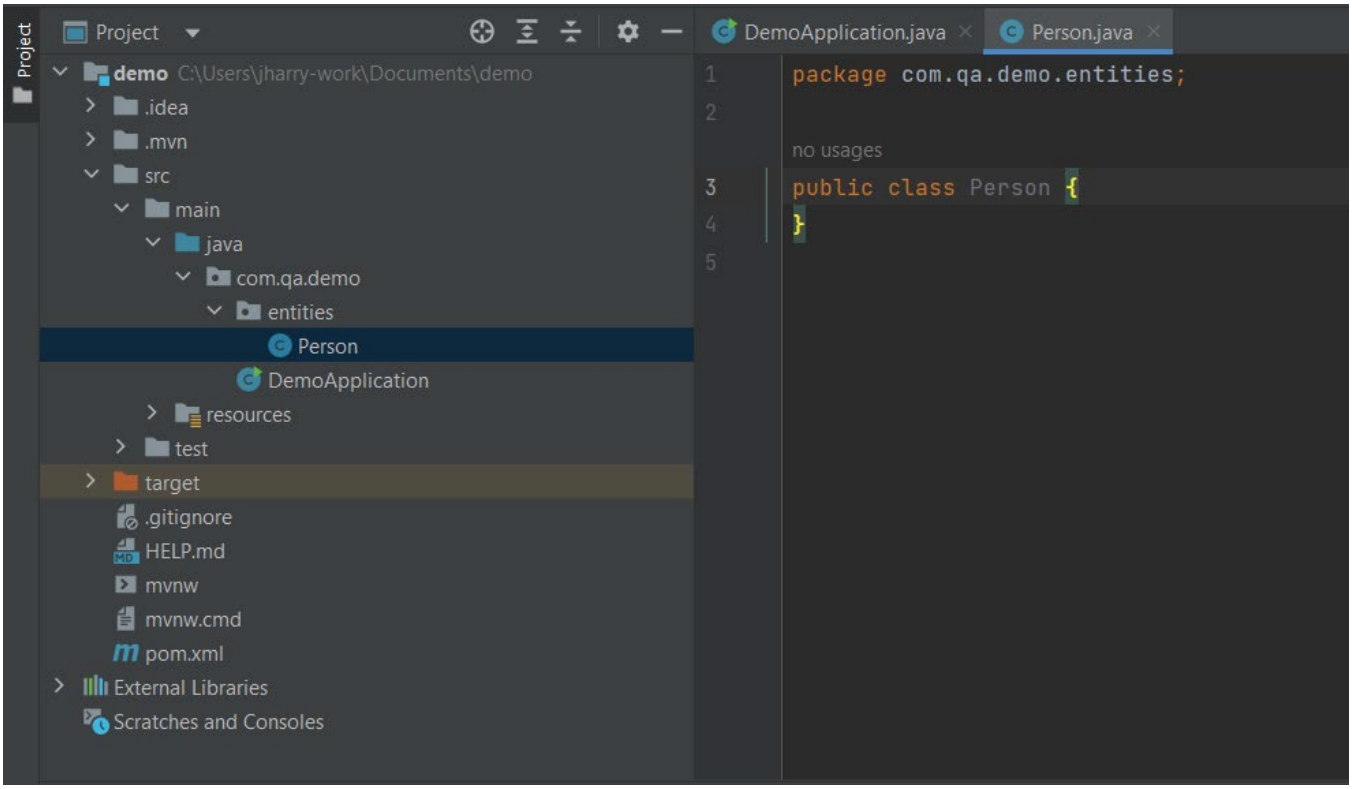
Step-by-step guide to create a new Spring Boot project:

1. Open the project you created in the previous lab.
2. Create a new package in **com.qa.demo** called **entities.**

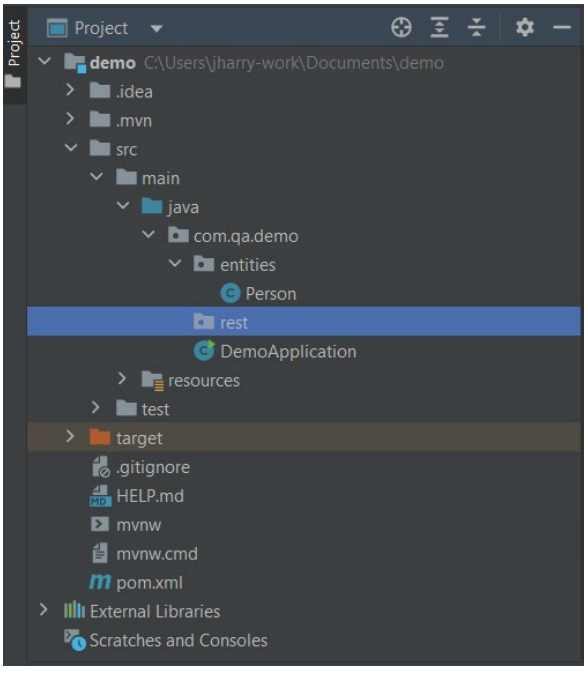


3. Inside the new package, create a class called Person.

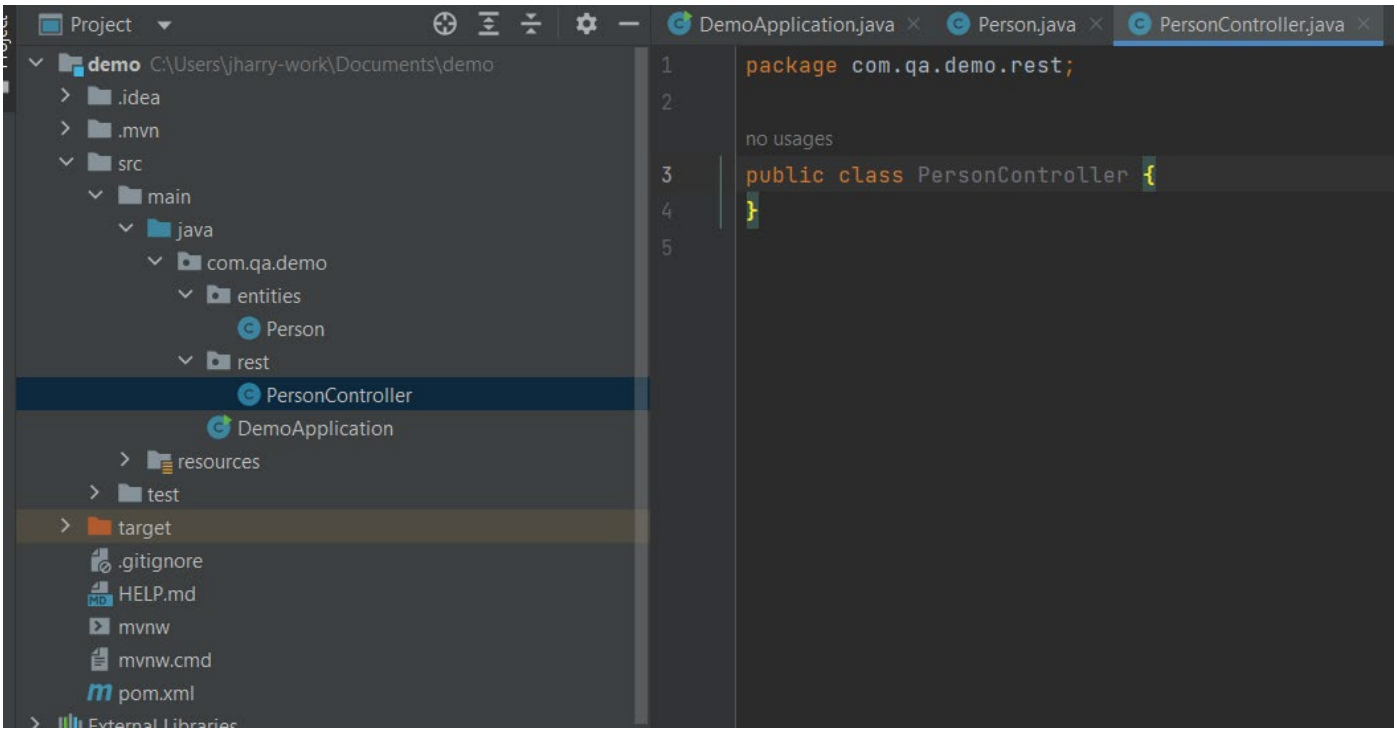
4. Add three properties to the Person class: name, age, and job. The class should be properly encapsulated with private properties, as well as public getters and setters.



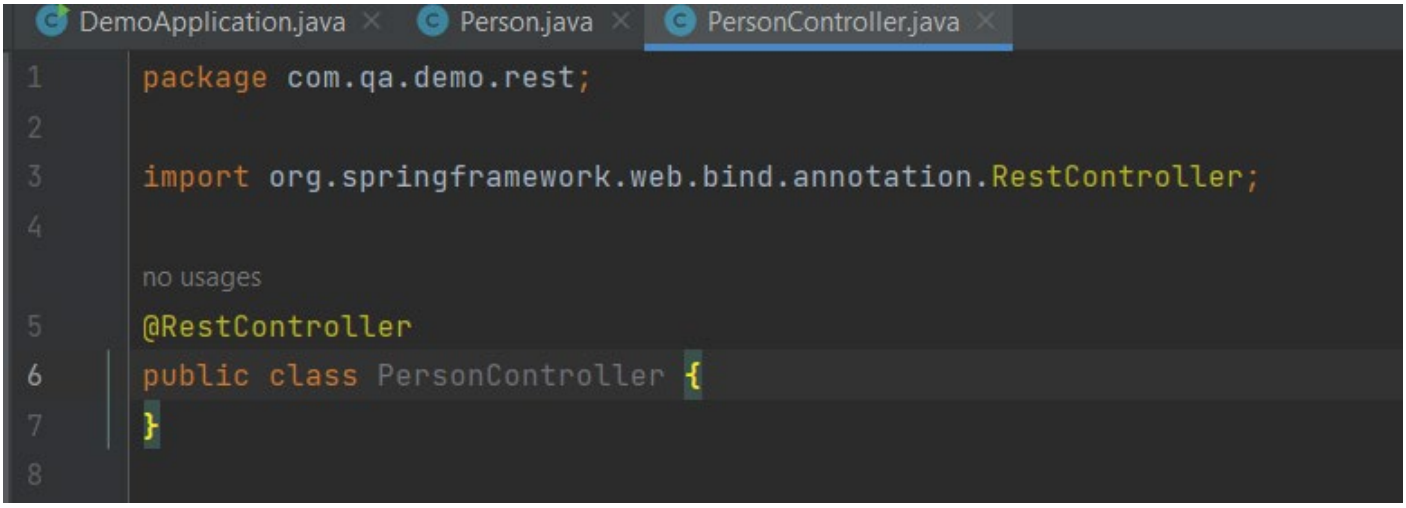
5. Create a new package in **com.qa.demo** called **rest.**



6. Add a **PersonController** class to the new package.



7. Add the **@RestController** annotation above the class declaration..



8. Add a List of Persons to the controller.

```
3   import com.qa.demo.entities.Person;
4   import org.springframework.web.bind.annotation.RestController;
5
6   import java.util.ArrayList;
7   import java.util.List;
8
    no usages
9   @RestController
10  public class PersonController {
11
        no usages
12      private List<Person> people = new ArrayList<>();
13  }
14
```

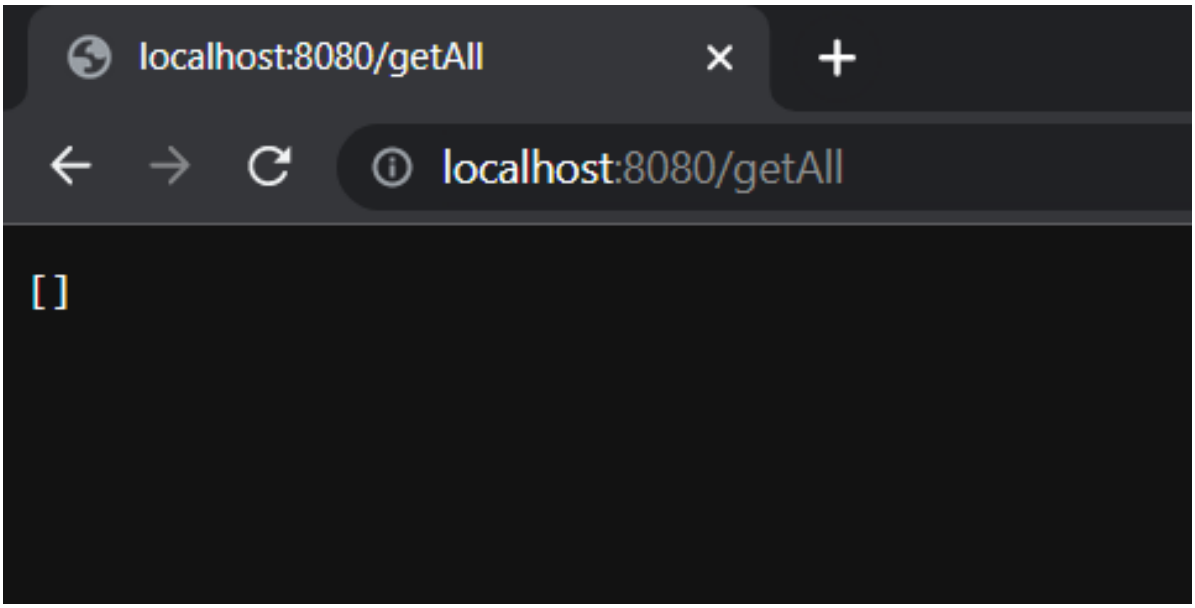
9. Add a **getAll** method that returns the **people** list.

```
no usages
@RestController
public class PersonController {

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

    no usages
    public List<Person> getAll() {
        return this.people;
    }
```

10. Annotate this method with **@GetMapping("/getAll")** – this will map the method to any **GET** requests Spring receives at the path **/getAll.**

```
no usages
@RestController
public class PersonController {

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

    no usages
    @GetMapping("/getAll")
    public List<Person> getAll() {
        return this.people;
    }
```

11. Test this endpoint by restarting your Spring app and going to http://localhost:8080/getAll - you should see an empty array.



12. Now that the array is accessable via the API, we can add some Person objects to it. Add a **createPerson** method with **Person** as the return type and first parameter.

```
no usages
public Person createPerson(@RequestBody Person person) {
    return null;
}
```

13. Add logic to the **createPerson** method so that it adds the new **person** to the **people** list.

```
no usages
public Person createPerson(Person person) {
    this.people.add(person);
    return this.people.get(this.people.size() - 1);
}
```

14. Map this method to a POST request at /create.

```
no usages
@PostMapping("/create")
public Person createPerson(Person person) {
    this.people.add(person);
    return this.people.get(this.people.size() - 1);
}
```

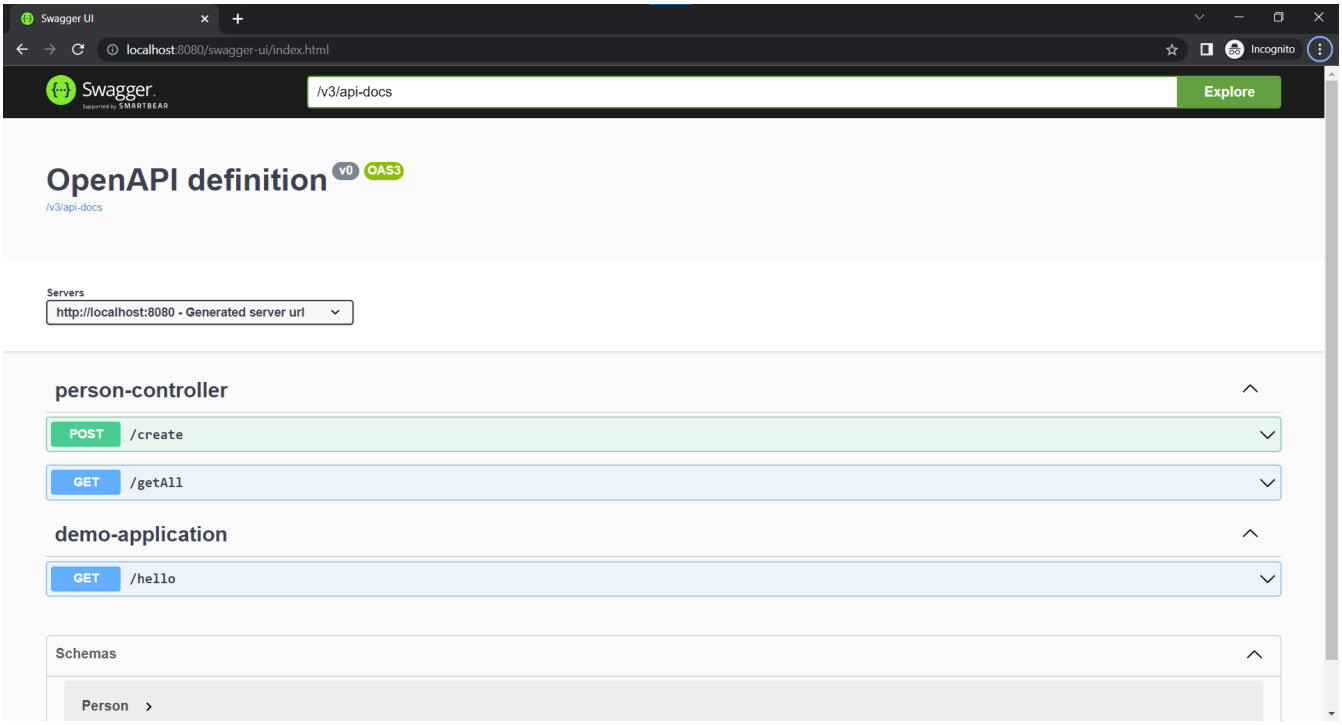
15. Because this method will be called via a POST request, the **person** data will be in the body of the request. Adding **@RequestBody** before the parameter will tell Spring to extract the **person** data from the body of the request.

```
no usages
@PostMapping("/create")
public Person createPerson(@RequestBody Person person) {
    this.people.add(person);
    return this.people.get(this.people.size() - 1);
}
```

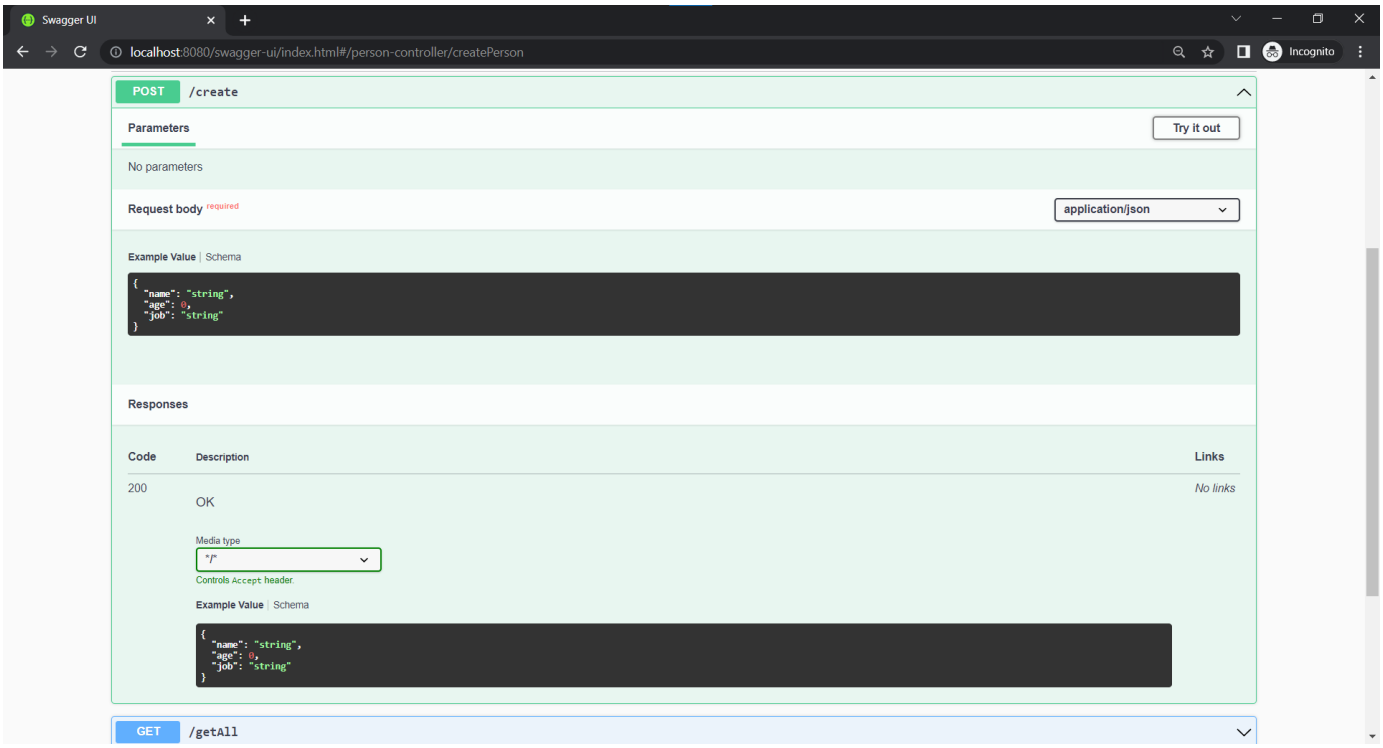
16. At this point, we would like to test our **createPerson** method but we can't use a browser to send a POST request, so we're going to need to add Swagger to our Spring project. Insert this snippet into the **dependencies** section of your POM file.

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.0.0</version>
</dependency>
```

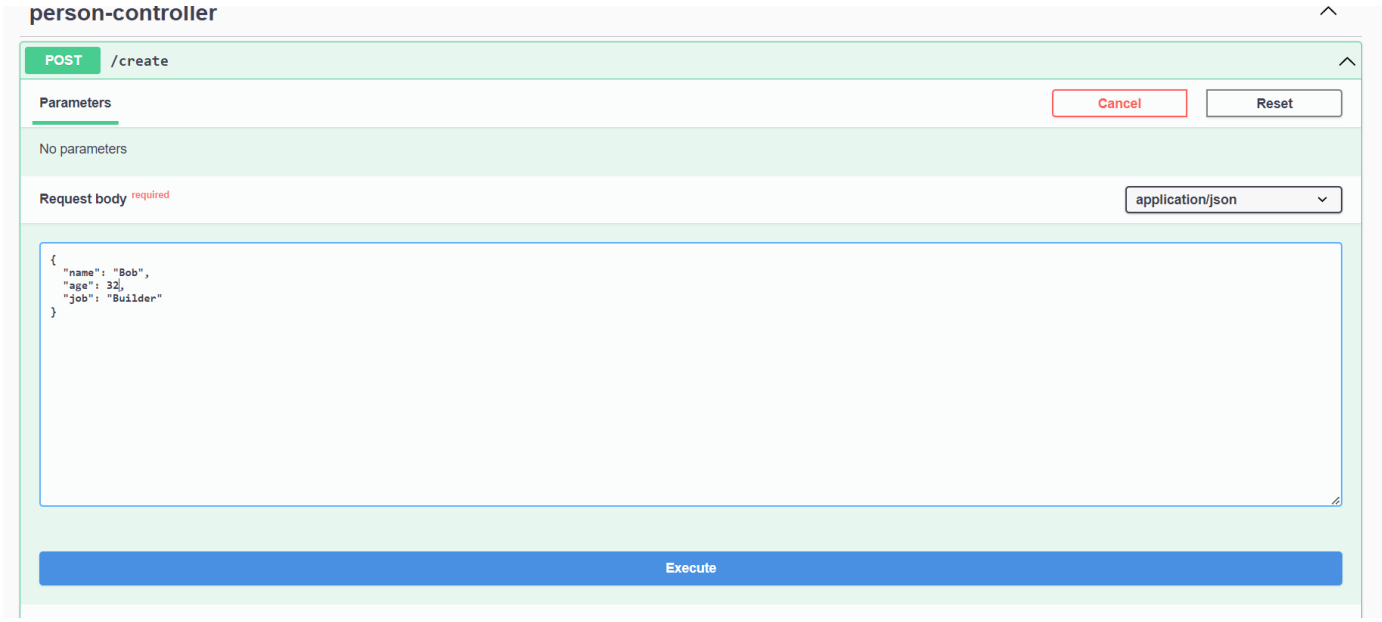
17. After restarting your app, you should now be able to go to http://localhost:8080/swagger-ui/index.html and see the generated docs for your API endpoints. Here we can see the create and get person methods, plus the 'Hello, World' endpoint from the previous lab.



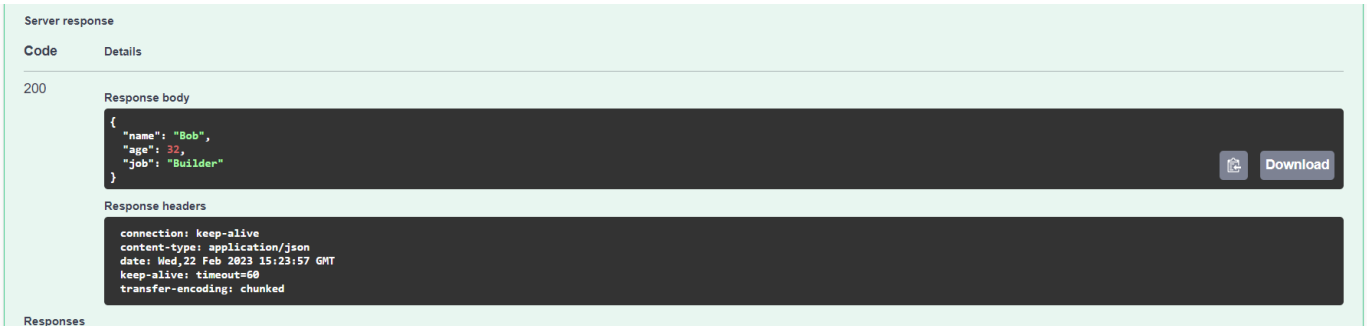
18. Take a look at the create endpoint.



19. From here we can see the structure of the input data the endpoint is expecting and also the structure of the output the endpoint will return. Use the **Try It Out** button to enter some test data.



20. Hit the **Execute** button and scroll down to see the response. You should see the data you entered in the previous step being returned in the response body, along with a 200 code.



21. Try to execute the **getAll** endpoint and you should see the same data in an array (because the endpoint returns a List<Person> rather than a Person object).

22. Go back to the **PersonController**. Add a **removePerson** method that returns a Person and accepts an **int id** (for now this 'id' is just going to be the index in the List, but in future labs it will be an actual id in the table). Add some logic to the method that removes the person at the index that is passed in, then returns the element that used to be at the index.

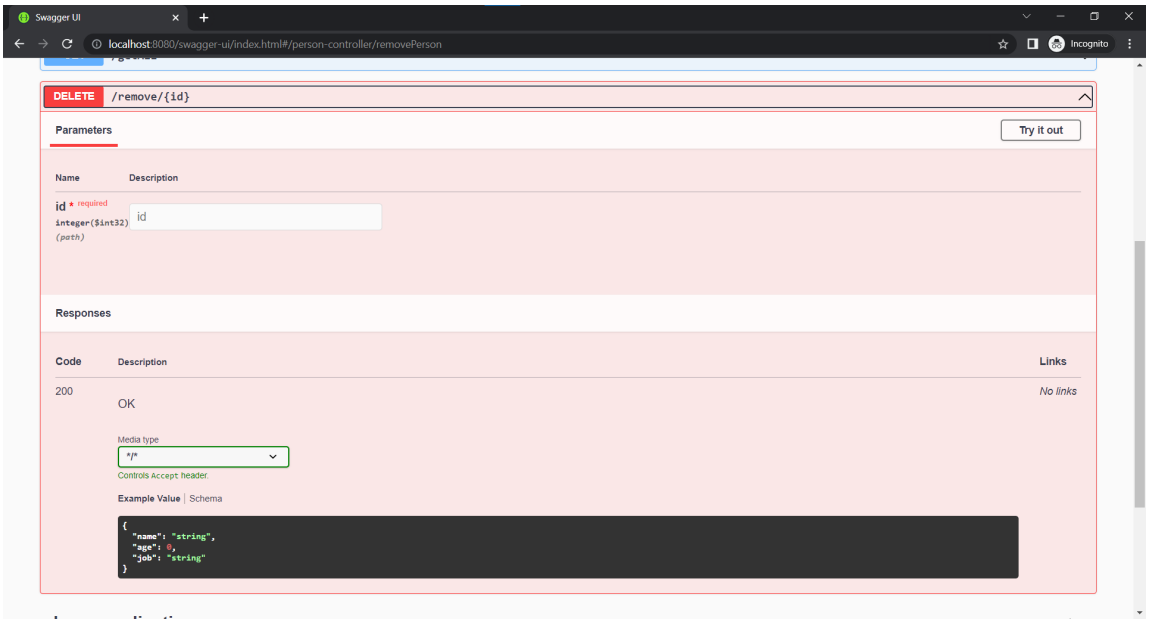
23. Map this method to a DELETE request at /remove.

```
    no usages
    @DeleteMapping("/remove")
    public Person removePerson(int id) {
        return this.people.remove(id);
    }
}
```

24. Now we need some way for the id to be passed through with the request, but using the body would be a bit overkill for a single int, so instead we're going to include it as part of the path. Add **/{id}** onto the end of the path and **@PathVariable** before the first parameter.

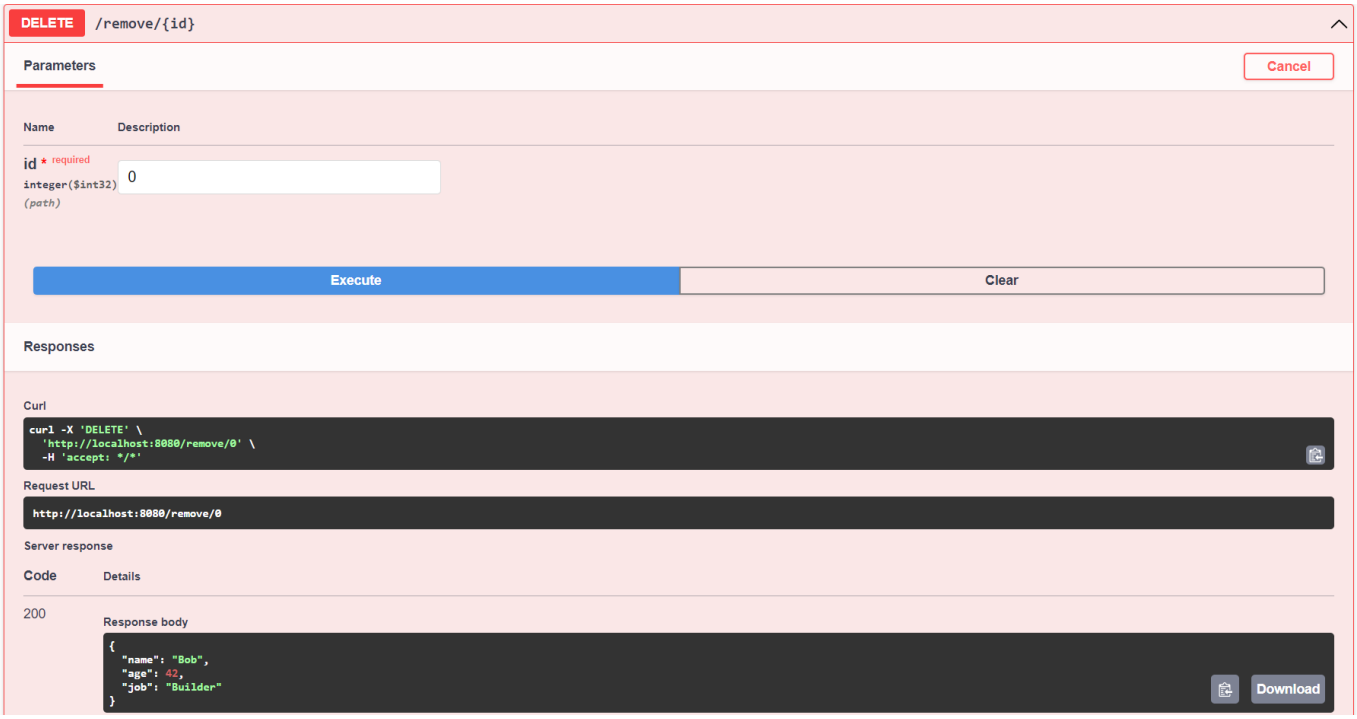
```
    no usages
    @DeleteMapping("/remove/{id}")
    public Person removePerson(@PathVariable int id) {
        return this.people.remove(id);
    }
}
```

25. Restart the app; go back to swagger docs and check the remove endpoint has appeared.

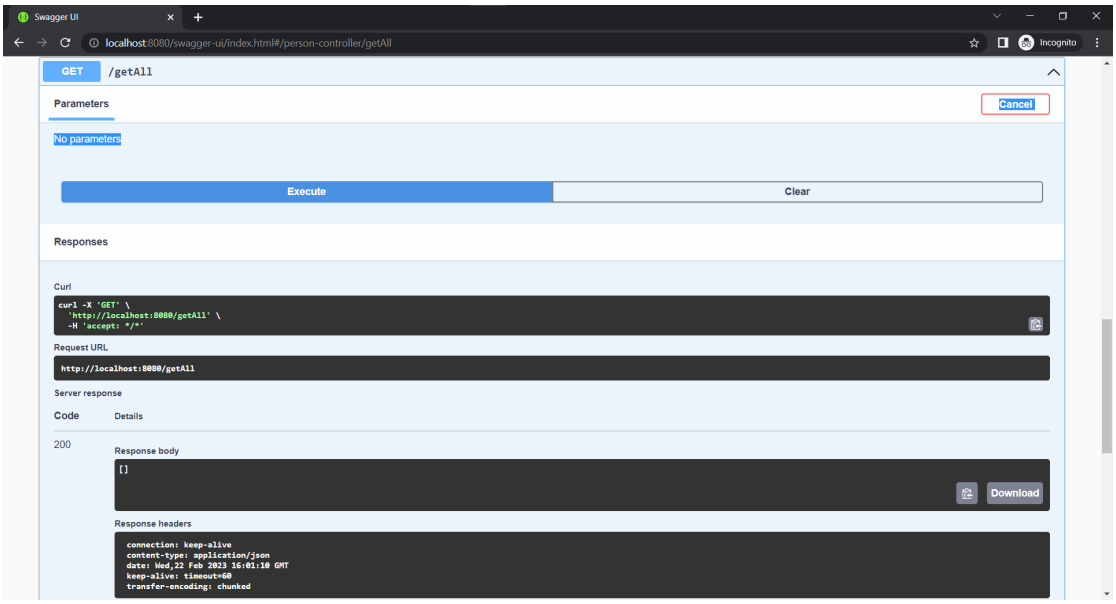


26. Use the create endpoint to add a person again, for something else to test the delete with.

27. Now click **Try It Out** with your remove endpoint and enter the “id” of the person you just created (it should be the only entry in the list so it’ll have an index of 0). You should see the same data from before in the response body.



28. Now if you try the getAll endpoint you will get an empty array, shoeing that the person has definitely been successfully deleted.



29. Add an **updatePerson** method to the **PersonController**. This method will return a Person and will take in 4 parameters: an int id, String name, Integer age and a String job.

```java
public Person updatePerson(int id, String name, Integer age, String job) {
    return null;
}
```

30. Add logic to the **updatePerson** method that finds the existing person at the specified index and sets the name, age and job to be equal to the passed in parameters (ignoring nulls).

```java
public Person updatePerson(int id, String name, Integer age, String job) {
    Person toUpdate = this.people.get(id);

    if (name != null) toUpdate.setName(name);
    if (age != null) toUpdate.setAge(age);
    if (job != null) toUpdate.setJob(job);

    return toUpdate;
}
```

31. Map this method to a PATCH request at /update.

```java
@PatchMapping("/update")
public Person updatePerson(int id, String name, Integer age, String job) {
    Person toUpdate = this.people.get(id);

    if (name != null) toUpdate.setName(name);
    if (age != null) toUpdate.setAge(age);
    if (job != null) toUpdate.setJob(job);

    return toUpdate;
}
```

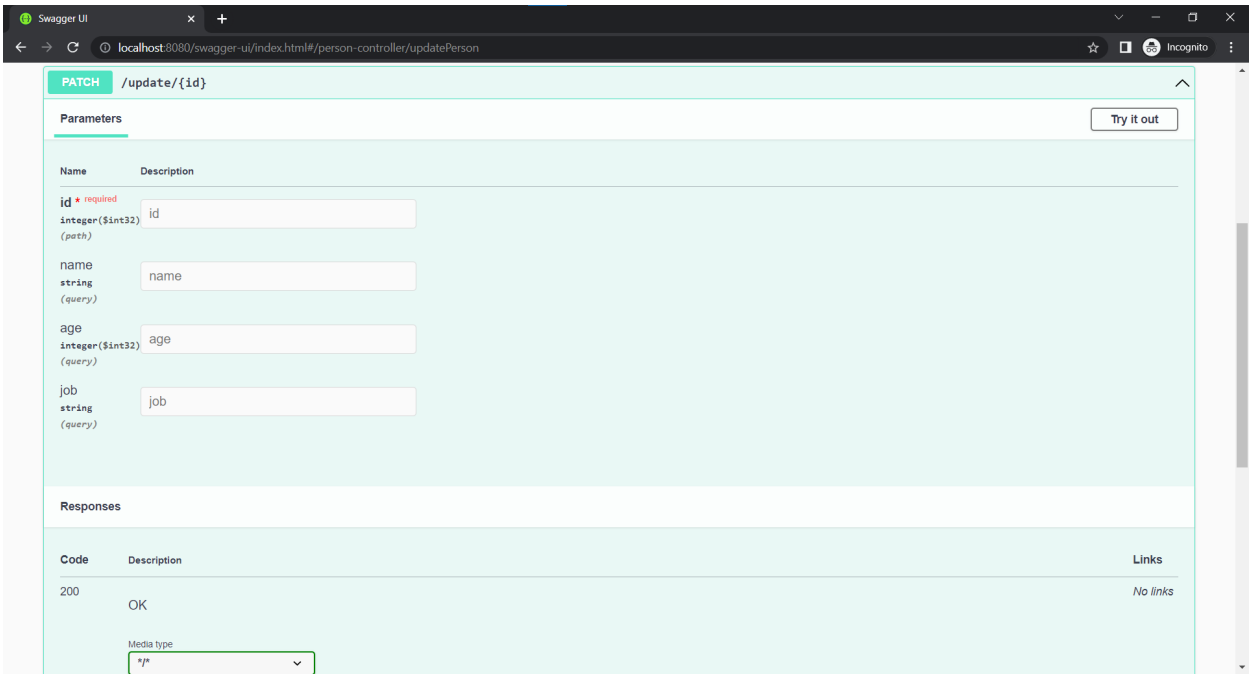
32. Make the **id** a **PathVariable** like you did for the remove method.

```
no usages
@PatchMapping("/update/{id}")
public Person updatePerson(@PathVariable int id, String name, Integer age, String job) {
    Person toUpdate = this.people.get(id);

    if (name != null) toUpdate.setName(name);
    if (age != null) toUpdate.setAge(age);
    if (job != null) toUpdate.setJob(job);

    return toUpdate;
}
```

33. We want the user to be able to pass in 1, 2, or 3 parameters, so we won't use either the request body or path variables; instead, we'll be using query parameters. Before the three remaining parameters, add the **@RequestParam** annotation with **required** set to **false**.

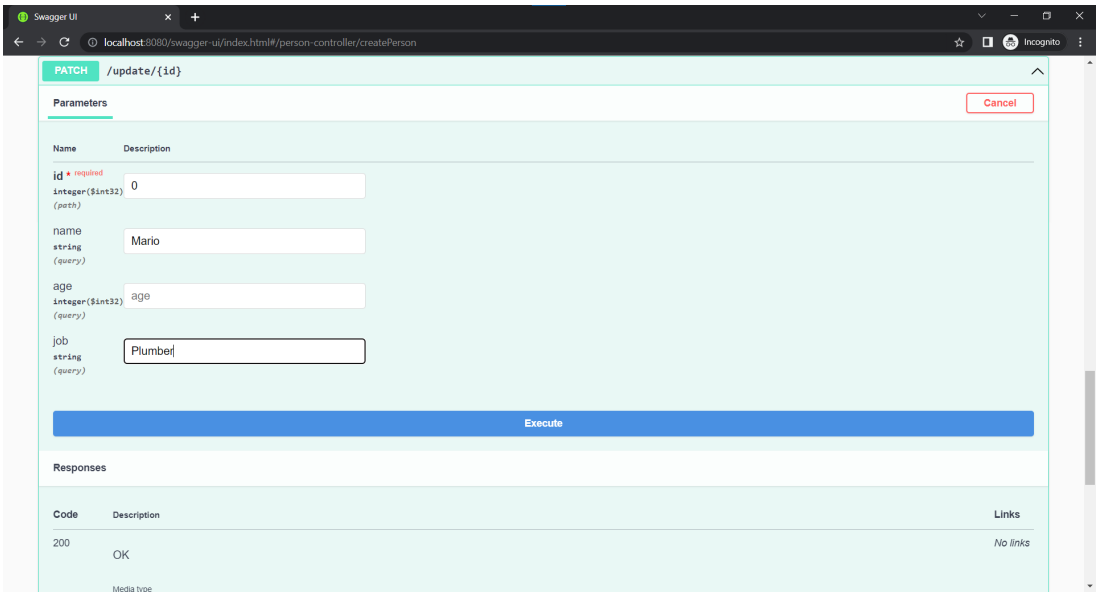
```
no usages
@PatchMapping("/update/{id}")
public Person updatePerson(@PathVariable int id,
                           @RequestParam(required = false) String name,
                           @RequestParam(required = false) Integer age,
                           @RequestParam(required = false) String job) {
    Person toUpdate = this.people.get(id);

    if (name != null) toUpdate.setName(name);
    if (age != null) toUpdate.setAge(age);
    if (job != null) toUpdate.setJob(job);

    return toUpdate;
}
```

34. Check the update method has been added to the Swagger docs and only the **id** is required.

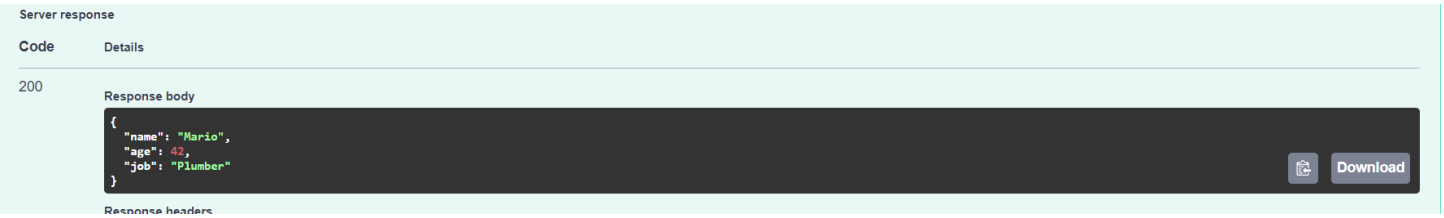


35. Add a test person using the create endpoint.

36. Use the update endpoint to send in new data – making sure to set the id to 0.



37. You should see the new name and job with the old age.



38. Finally, have a go at implementing a getById method in the **PersonController** that finds a *single* person using the 'id'.