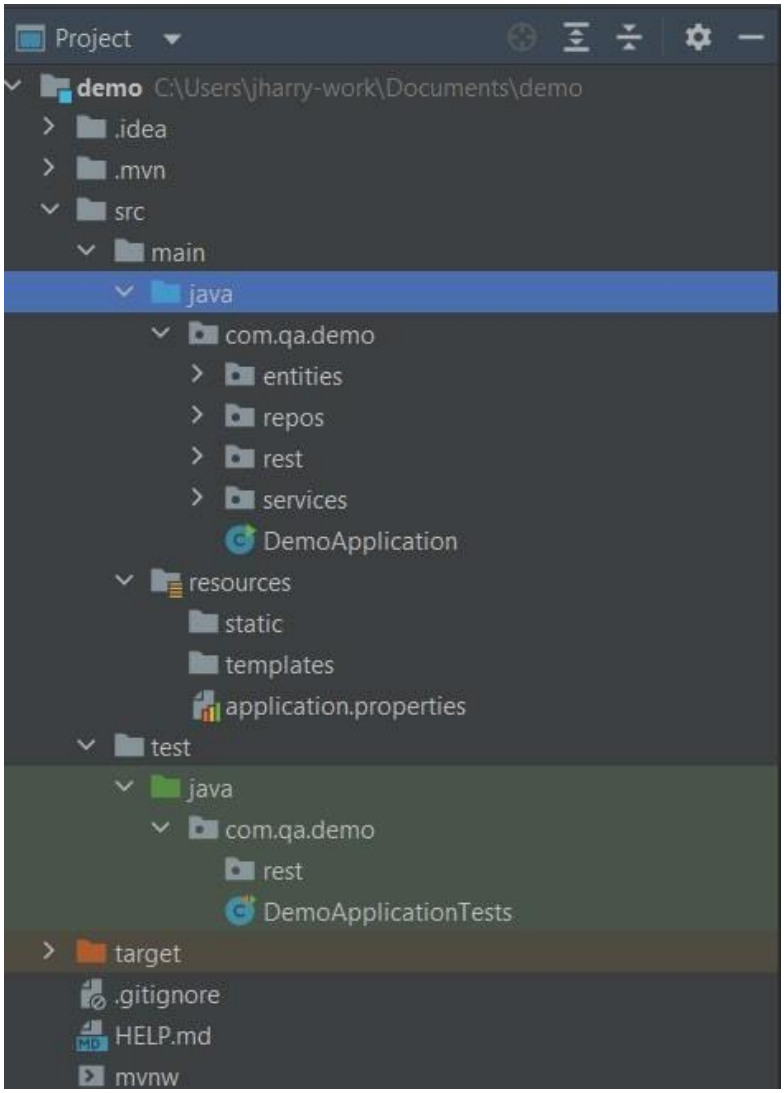
Lab 5 – Testing

MockMvc is a library used to send mock requests from our tests to the endpoint we created in previous labs. In this task, we will use MockMvc to test the create method from our PersonController.

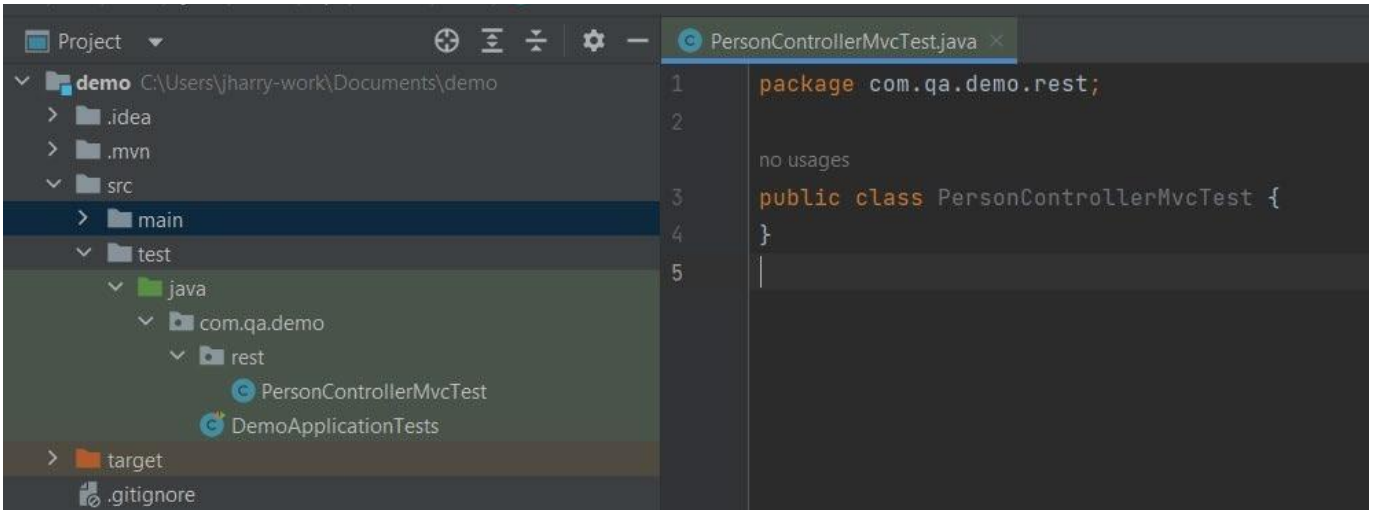
Task 1

1. Open the project you created in the previous lab.

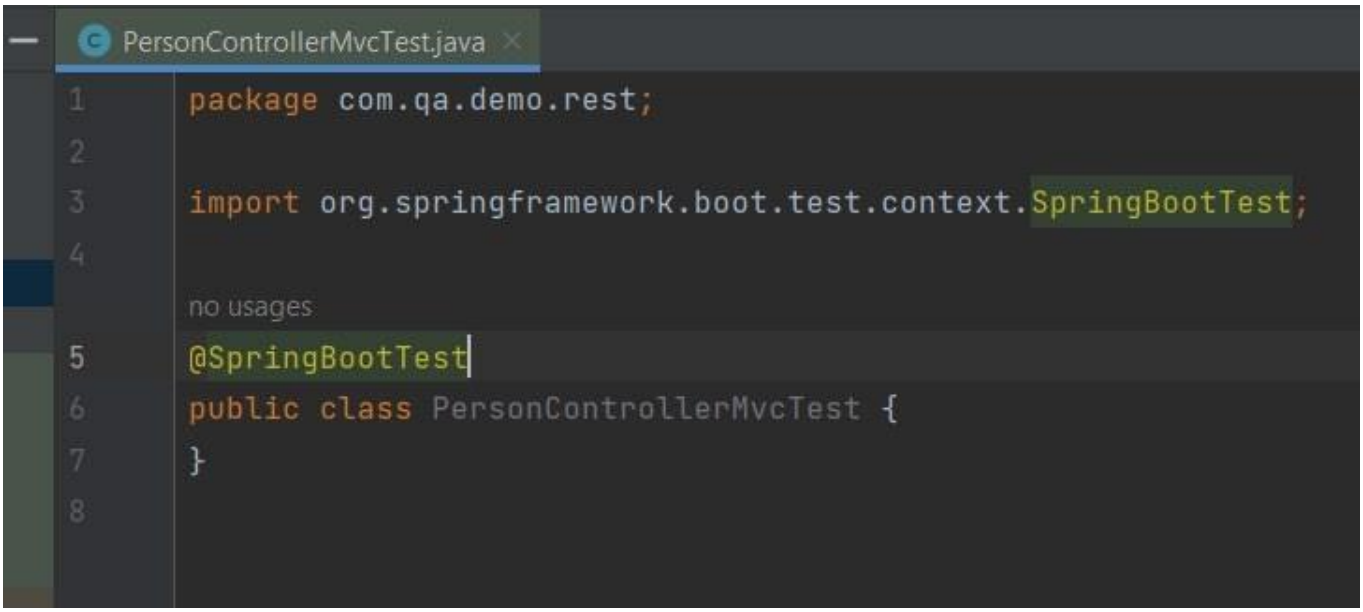
2. In **src/test/java** create a new package, **com.qa.demo.rest**. With Spring testing, it helps to have the same package structure in the testing folder as in main, because it lets the tests automatically discover the Spring context.



3. Within this new package, create a class called **PersonControllerMvcTest.**



4. Annotate the class as a **@SpringBootTest**, this will allow it to access the Spring context which is required for performing any kind of integration test.



5. Spring uses Junit 5 for its tests, so create a package-private **testCreate** method using **@Test** from the Jupiter API.

```java
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

no usages
@SpringBootTest
public class PersonControllerMvcTest {

    no usages
    @Test
    void testCreate() {

    }
}
```

6. Inject an instance of the **MockMVC** class, this Is the class you'll use to perform the mock requests and thus drives a significant portion of Spring testing.

```java
no usages
@SpringBootTest
public class PersonControllerMvcTest {

    no usages
    @Autowired
    private MockMvc mvc;

    no usages
    @Test
    void testCreate() {

    }
}
```

7. We can get Spring to configure this **MockMvc** object for us using **@AutoConfigureMockMvc**.

8. In order to create a mock request, we will need an instance of **RequestBuilder**.

9. Create a POST request using the **post** method from **MockMvcRequestBuilders** and set the path to /create.

```
no usages
@Test
void testCreate() {

    RequestBuilder mockRequest = MockMvcRequestBuilders.post( urlTemplate: "/create");

}
```

10. The **createPerson** method is set up to expect JSON data, so we need to set the content-type to application/json

```
@Test
void testCreate() {

    RequestBuilder mockRequest = MockMvcRequestBuilders.post( urlTemplate: "/create").contentType(MediaType.APPLICATION_JSON);

}
```

11. The final part of the request we need to set is the body. This can be done using the **content** method, but first we need to create the JSON string. Start by creating an example Person object.

```
@Test
void testCreate() {
    Person newPerson = new Person( name: "Bob",  age: 42,  job: "Builder");
    RequestBuilder mockRequest = MockMvcRequestBuilders.post( urlTemplate: "/

}
```

12. To convert **newPerson** into JSON we can use the **ObjectMapper** provided by Spring.

```
no usages
@SpringBootTest
@AutoConfigureMockMvc
public class PersonControllerMvcTest {

    no usages
    @Autowired
    private MockMvc mvc;

    no usages
    @Autowired
    private ObjectMapper mapper;

    no usages
    @Test
    void testCreate() {
        Person newPerson = new Person( name: "Bob", age: 42, job: "Builder");
```

13. Now use the **writeValuesAString** method to convert **newPerson** to JSON (Note that this method throws a checked exception so you will need to add **throws Exception** to the method declaration).

```
no usages
@Test
void testCreate() throws Exception {
    Person newPerson = new Person( name: "Bob", age: 42, job: "Builder");
    String newPersonAsJson = this.mapper.writeValueAsString(newPerson);
    RequestBuilder mockRequest = MockMvcRequestBuilders.post( urlTemplate: "/create")

}
```

14. Final part of building the request – insert the **newPersonAsJson** into the **content** method.

```
@Test
void testCreate() throws Exception {
    Person newPerson = new Person( name: "Bob", age: 42, job: "Builder");
    String newPersonAsJson = this.mapper.writeValueAsString(newPerson);
    RequestBuilder mockRequest = MockMvcRequestBuilders.post( urlTemplate: "/create").contentType(MediaType.APPLICATION_JSON).content(newPersonAsJson);

}
```

15. In order to test the response we will need two instances of **ResultMatcher**; one to test the status code and one to check the body.

```
@Test
void testCreate() throws Exception {
    Person newPerson = new Person( name
    String newPersonAsJson = this.mappe
    RequestBuilder mockRequest = MockMv

    ResultMatcher checkStatus;

    ResultMatcher checkBody;
}
```

16. To check the status code use the **status** method from **MockMvcResultMatchers**.

```
no usages
@Test
void testCreate() throws Exception {
    Person newPerson = new Person( name: "Bob",  age: 42,  job: "Builder");
    String newPersonAsJson = this.mapper.writeValueAsString(newPerson);
    RequestBuilder mockRequest = MockMvcRequestBuilders.post( urlTemplate:

    ResultMatcher checkStatus = MockMvcResultMatchers.status().isOk();

    ResultMatcher checkBody;
}
```

17. Before checking the body, we will need a JSON string to compare it to. Create another test person with the same data as the first one except it has an id of 1, then use the **ObjectMapper** to convert it to JSON.

```
@Test
void testCreate() throws Exception {
    Person newPerson = new Person( name: "Bob",  age: 42,  job: "Builder");
    String newPersonAsJson = this.mapper.writeValueAsString(newPerson);
    RequestBuilder mockRequest = MockMvcRequestBuilders.post( urlTemplate: "/create")

    ResultMatcher checkStatus = MockMvcResultMatchers.status().isOk();
    Person createdPerson = new Person( id: 1, name: "Bob",  age: 42,  job: "Builder");
    String createdPersonAsJson = this.mapper.writeValueAsString(createdPerson);
    ResultMatcher checkBody;
}
```

18. Now to check the body use the **content** method from **MockMvcResultMatchers**.

```
no usages
@Test
void testCreate() throws Exception {
    Person newPerson = new Person( name: "Bob",  age: 42,  job: "Builder");
    String newPersonAsJson = this.mapper.writeValueAsString(newPerson);
    RequestBuilder mockRequest = MockMvcRequestBuilders.post( urlTemplate: "/create").contentTyp

    ResultMatcher checkStatus = MockMvcResultMatchers.status().isOk();
    Person createdPerson = new Person( id: 1, name: "Bob",  age: 42,  job: "Builder");
    String createdPersonAsJson = this.mapper.writeValueAsString(createdPerson);
    ResultMatcher checkBody = MockMvcResultMatchers.content().json(createdPersonAsJson);
}
```

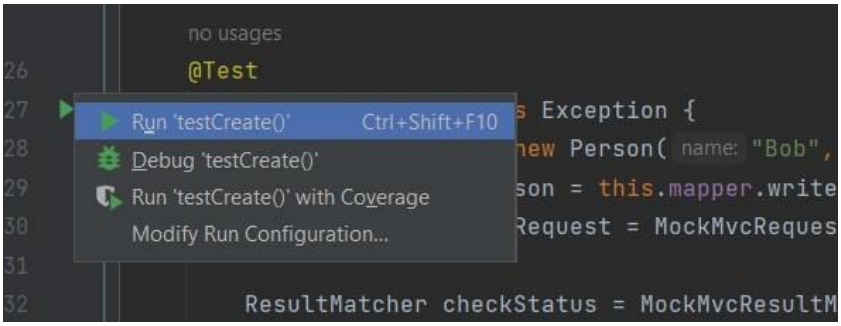
19. Send the mock request using the **perform** method from our **MockMvc** object.

```
@Test
void testCreate() throws Exception {
    Person newPerson = new Person( name: "Bob",  age: 42,  job: "Builder");
    String newPersonAsJson = this.mapper.writeValueAsString(newPerson);
    RequestBuilder mockRequest = MockMvcRequestBuilders.post( urlTemplate: "/create").contentT

    ResultMatcher checkStatus = MockMvcResultMatchers.status().isOk();
    Person createdPerson = new Person( id: 1, name: "Bob",  age: 42,  job: "Builder");
    String createdPersonAsJson = this.mapper.writeValueAsString(createdPerson);
    ResultMatcher checkBody = MockMvcResultMatchers.content().json(createdPersonAsJson);

    this.mvc.perform(mockRequest);
}
```

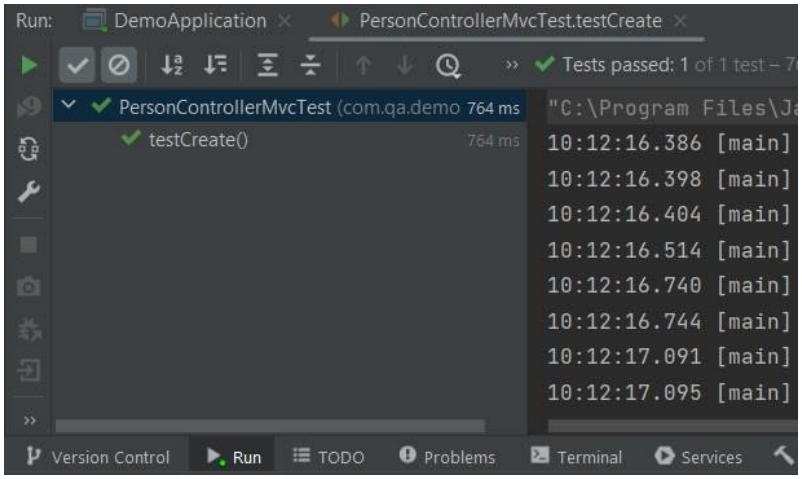
20. Finally, chain a couple **andExpect** methods to run the checks we just created.

```
no usages
@Test
void testCreate() throws Exception {
    Person newPerson = new Person( name: "Bob",  age: 42,  job: "Builder");
    String newPersonAsJson = this.mapper.writeValueAsString(newPerson);
    RequestBuilder mockRequest = MockMvcRequestBuilders.post( urlTemplate: "/create").content

    ResultMatcher checkStatus = MockMvcResultMatchers.status().isOk();
    Person createdPerson = new Person( id: 1, name: "Bob",  age: 42,  job: "Builder");
    String createdPersonAsJson = this.mapper.writeValueAsString(createdPerson);
    ResultMatcher checkBody = MockMvcResultMatchers.content().json(createdPersonAsJson);

    this.mvc.perform(mockRequest).andExpect(checkStatus).andExpect(checkBody);
}
```

21. Now we can run the test from the gutter:



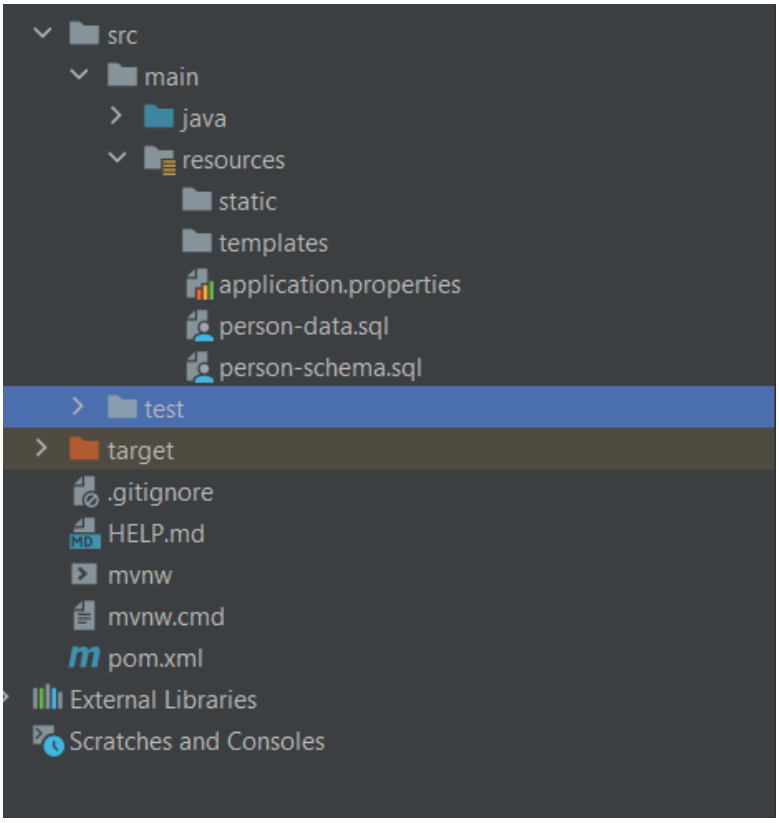
22. And check that the test passes:



Task 2

When performing multiple tests it can be beneficial to reset the database between each test to ensure a consistent database state to test against. In Spring this can be accomplished with the @Sql annotation.

1. Go to your project from the previous task: In **src/main/resources** add two files; **person-schema.sql** and **person-data.sql**. These files will be used to create and populate the tables for our tests.



2. In the schema file we will add the sql to create the **person** table. (Note: this MUST match the sql that is auto-generated by Spring from the entity class, otherwise the tests when fail when attempting to access or modify the table. Please refer back to the Person class if you're unsure of the purpose of any part of the following queries)

```
DROP TABLE IF EXISTS `person`;
CREATE TABLE `person` (
    `id` INTEGER PRIMARY KEY AUTO_INCREMENT,
    `full_name` VARCHAR NOT NULL UNIQUE,
    `age` INTEGER,
    `job` VARCHAR
);
```

3. Use **@Sql** to import and execute the scripts in the **PersonControllerMvcTest** class; remembering to run the scripts *before* the tests and making sure the schema file is executed *first*.

```
no usages
@SpringBootTest
@AutoConfigureMockMvc
@Sql(executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD,
        scripts = {"classpath:person-schema.sql", "classpath:person-data.sql"})
public class PersonControllerMvcTest {
```

4. Now that we have a populated database we can do more than just testing the ability to create new records. Using what you've learned from writing the create test try and write at least one test for each of the **get**, **getAll**, **updatePerson** and **deletePerson** methods. (Solutions are on the following page if you get stuck)

Task 2 - Solutions

Get Person

```
no usages
@Test
void testRead() throws Exception {
    int id = 1;
    RequestBuilder mockRequest = MockMvcRequestBuilders.get( urlTemplate: "/get/" + id);

    ResultMatcher checkStatus = MockMvcResultMatchers.status().isOk();

    Person existing = new Person(id, name: "Pat", age: 53, job: "Postman");
    String existingPersonAsJson  = this.mapper.writeValueAsString(existing);
    ResultMatcher checkBody = MockMvcResultMatchers.content().json(existingPersonAsJson);

    this.mvc.perform(mockRequest).andExpect(checkStatus).andExpect(checkBody);
}
```

Update Person

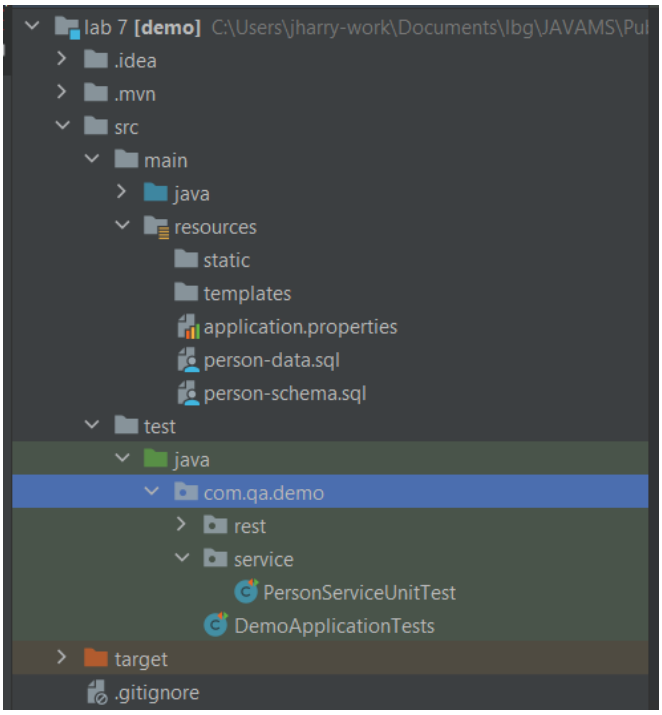
```
no usages
@Test
void testUpdate() throws Exception{
    int id = 1;
    Person updated = new Person(id, name: "Top", age: 50, job: "Cat");

    RequestBuilder mockRequest = MockMvcRequestBuilders.patch( urlTemplate: "/update/" + id)
            .queryParam( name: "name", updated.getName())
            .queryParam( name: "age", String.valueOf(updated.getAge()))
            .queryParam( name: "job", updated.getJob());

    ResultMatcher checkStatus = MockMvcResultMatchers.status().isOk();

    String updatedPersonAsJson  = this.mapper.writeValueAsString(updated);
    ResultMatcher checkBody = MockMvcResultMatchers.content().json(updatedPersonAsJson);

    this.mvc.perform(mockRequest).andExpect(checkStatus).andExpect(checkBody);
}
```

Delete Person

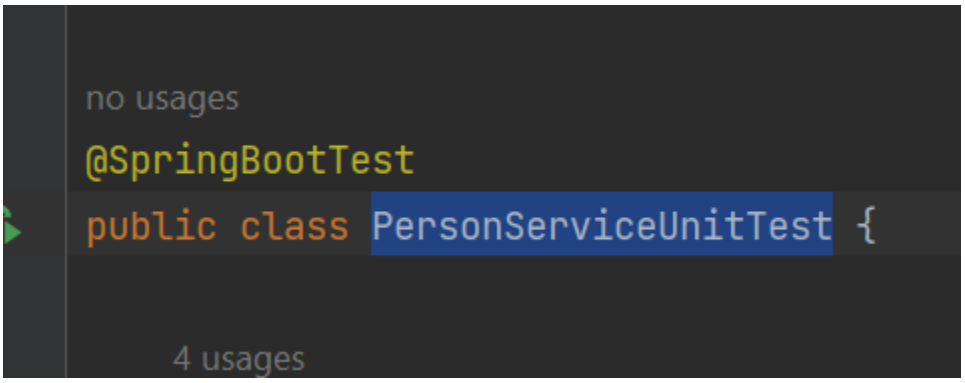
```
no usages
@Test
void testDelete() throws Exception{
    int id = 1;
    RequestBuilder mockRequest = MockMvcRequestBuilders.delete( urlTemplate: "/remove/" + id);

    ResultMatcher checkStatus = MockMvcResultMatchers.status().isOk();

    Person existing = new Person(id, name: "Pat", age: 53, job: "Postman");
    String existingPersonAsJson  = this.mapper.writeValueAsString(existing);
    ResultMatcher checkBody = MockMvcResultMatchers.content().json(existingPersonAsJson);

    this.mvc.perform(mockRequest).andExpect(checkStatus).andExpect(checkBody);
}
```

Task 3

1. Create a new package in **src/main/test** called **service** with a new test class called **PersonServiceUnitTest.**



2. Add the **@SpringBootTest** annotation to the new test class



3. Tell Spring to autowire in an istance of the class we're going to be testing (**PersonService**).

```
no usages
@SpringBootTest
public class PersonServiceUnitTest {

    4 usages
    @Autowired
    private PersonService service;
```

4. Because this is a *unit* test we want to look at the **PersonService** in isolation. To facilitate this we should mock any dependencies of the **PersonService** class to prevent them interfering with our test of the **PersonService** logic. We can instruct Spring to create a mock version of any bean using **@MockBean**.

```
no usages
@SpringBootTest
public class PersonServiceUnitTest {

    4 usages
    @Autowired
    private PersonService service;

    5 usages
    @MockBean
    private PersonRepo repo;
```

5. We'll start with testing the **updatePerson** method as it's the easiest to see the Mockito functionality in action. First, create a new test called **testUpdate**.

```
no usages
@Test
void testUpdate() {

}
```

6. The first step in most tests is creating the test data. We will start with creating a test Person – this will serve two functions: First, it will be used as the input parameters for the method being tested. Second, it will *also* be used to provide the return value of the **save** method of the **PersonRepo**. This test data is used twice because with an update method you will expect the data returned (the updated entity) to be largely the same as the the input (the new data used to update the existing entity).

```
no usages
@Test
void testUpdate() {
    // GIVEN
    int id = 1;
    Person updated = new Person(id, name: "Top", age: 50, job: "Cat");

    //WHEN

    // THEN
}
```

7. Next step is to create the mocks; the general rule with Mockito is you will want to create a mock for each call to a mocked bean in the method being tested. In the case of the **update** method there are two calls to the PersonRepo; one to the **save** method and one to the **findById** method. For this test we will force the **findById** method to return any Person object (as all of the properties will be overriden during the update it doesn't matter what the **name**, **age**

and **job** are at the moment) and the **save** method to return the test Person created in the previous step.

```
no usages
@Test
void testUpdate() {
    // GIVEN
    int id = 1;
    Person updated = new Person(id, name: "Top", age: 50, job: "Cat");

    //WHEN
    Mockito.when(this.repo.findById(id)).thenReturn(Optional.of(new Person( id: 1, name: null, age: null, job: null)));
    Mockito.when(this.repo.save(updated)).thenReturn(updated);

    // THEN
}
```

8. When using Mockito it is very important to remember to provide an **equals** method to any class used as input for a mock (**Person**, in this case). This is because Mockito uses the **equals** method to check whether the data it has been provided matches the data it has been told to expect so even if you provide an identical object to the one it's expecting it'll fail the simple **==** check Java performs by default and you'll merely get **null** returned rather than the data you were expecting.
   Now use IntelliJ to generate an **equals** method in **Person**.
9. The final step is to perform our assertion – here we'll use **JUnit** to check to see if the **update** method returns our expected value

```
no usages
@Test
void testUpdate() {
    // GIVEN
    int id = 1;
    Person updated = new Person(id, name: "Top", age: 50, job: "Cat");

    //WHEN
    Mockito.when(this.repo.findById(id)).thenReturn(Optional.of(new Person( id: 1, name: null, age: null, job: null)));
    Mockito.when(this.repo.save(updated)).thenReturn(updated);

    // THEN
    Assertions.assertEquals(updated,
            this.service.updatePerson(id, updated.getName(), updated.getAge(), updated.getJob()));
}
```

10. Now that we've done one Mockito test try and write at least one test for each of the other methods in **PersonService** (solutions are available on the next page if required) – remember you do *not* need to mock a void method!

Task 2 – Solutions

Create Person

```
no usages
@Test
void testCreate() {
    Person newPerson = new Person( name: "Bill",  age: 27,  job: "Accountant");
    Person savedPerson = new Person( id: 1, name: "Bill",  age: 27,  job: "Accountant");

    Mockito.when(this.repo.save(newPerson)).thenReturn(savedPerson);

    Assertions.assertEquals(savedPerson, this.service.createPerson(newPerson));
}
```

Get Person

```
no usages
@Test
void testGet() {
    int id = 1;
    Person existingPerson = new Person(id, name: "Ben",  age: 27,  job: "Actuary");

    Mockito.when(this.repo.findById(id)).thenReturn(Optional.of(existingPerson));

    Assertions.assertEquals(existingPerson, this.service.get(id));
}
```

Delete Person

```
no usages
@Test
void testDelete() {
    int id = 1;
    Person existingPerson = new Person(id, name: "Ben",  age: 27,  job: "Actuary");

    Mockito.when(this.repo.findById(id)).thenReturn(Optional.of(existingPerson));

    Assertions.assertEquals(existingPerson, this.service.removePerson(id));
}
```