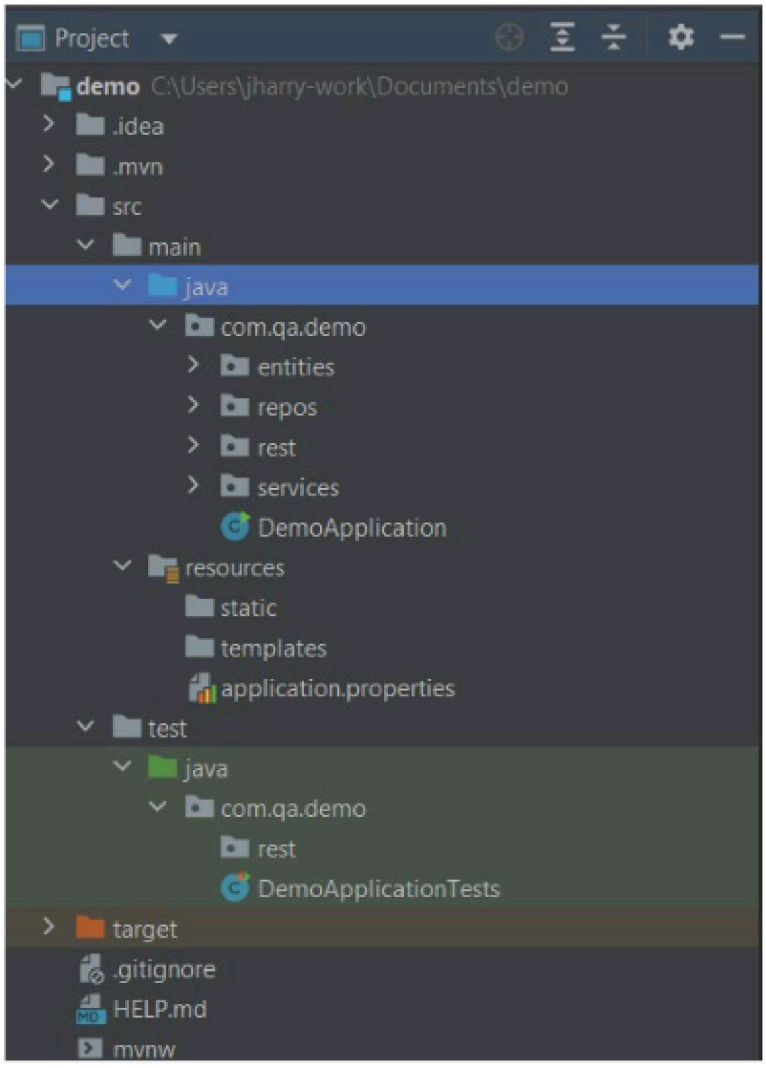
Lab 5 – Testing

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.

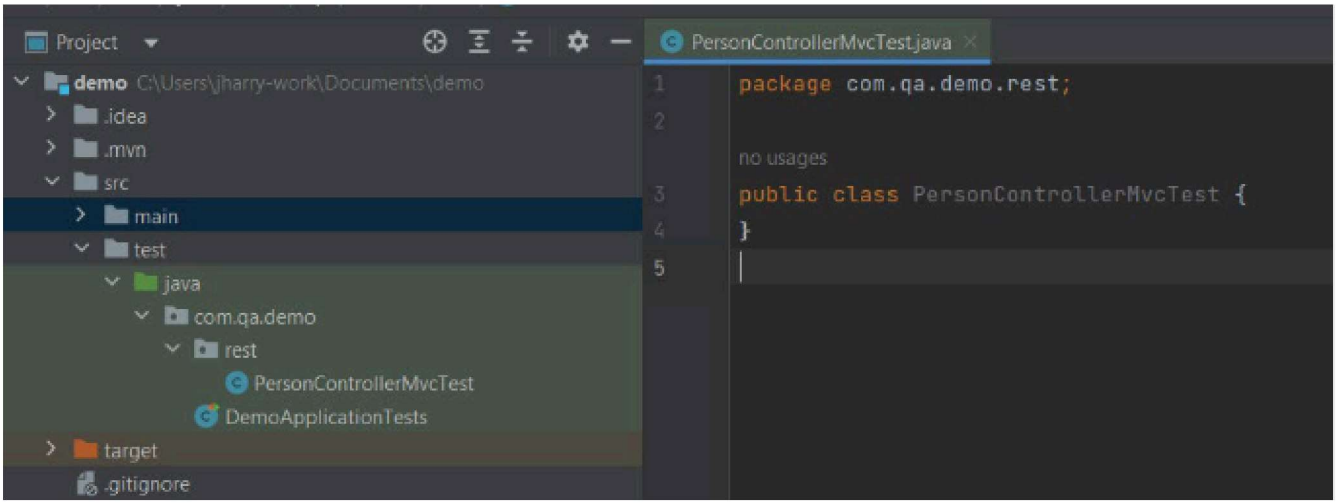
Task 1

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

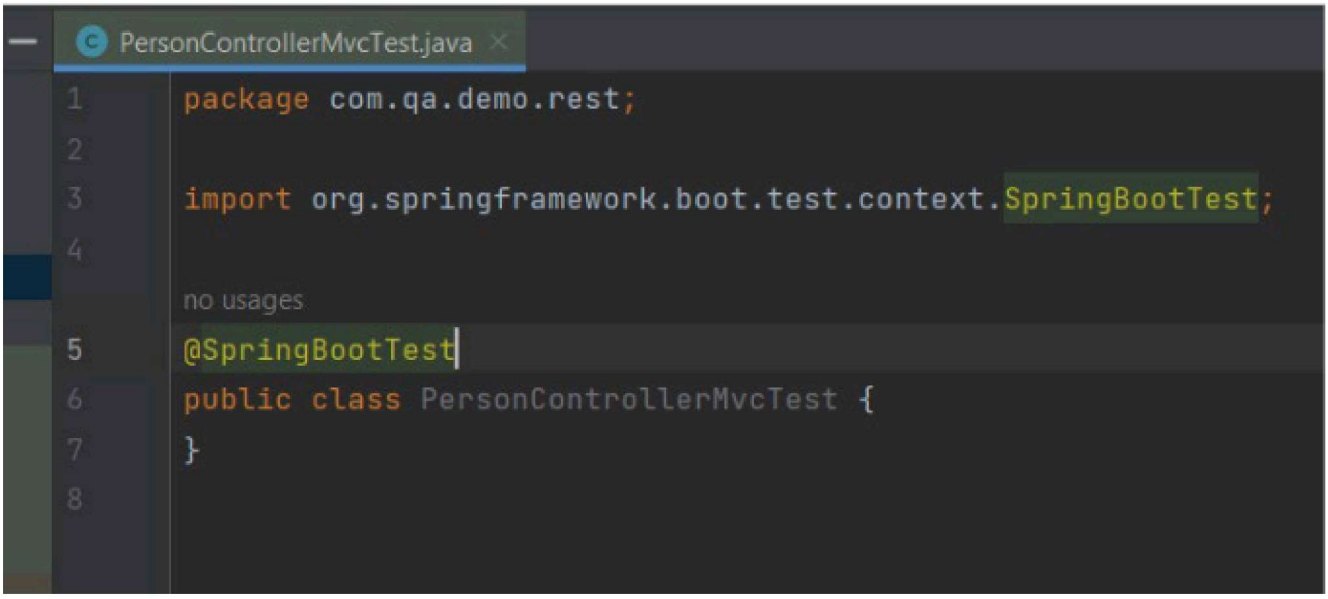
2. In **src/test/java** create a new package, **com.qa.demo.rest**. With Spring testing, it helps to have the same package structure in the testing folder as in main, because it lets the tests automatically discover the Spring context.



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.

```
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.

```
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**.

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

    no usages
    @Autowired
    private MockMvc mvc;

    no usages
    @Test
    void testCreate() {

    }
}
```

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

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

    RequestBuilder mockRequest;

}
```

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");

}
```

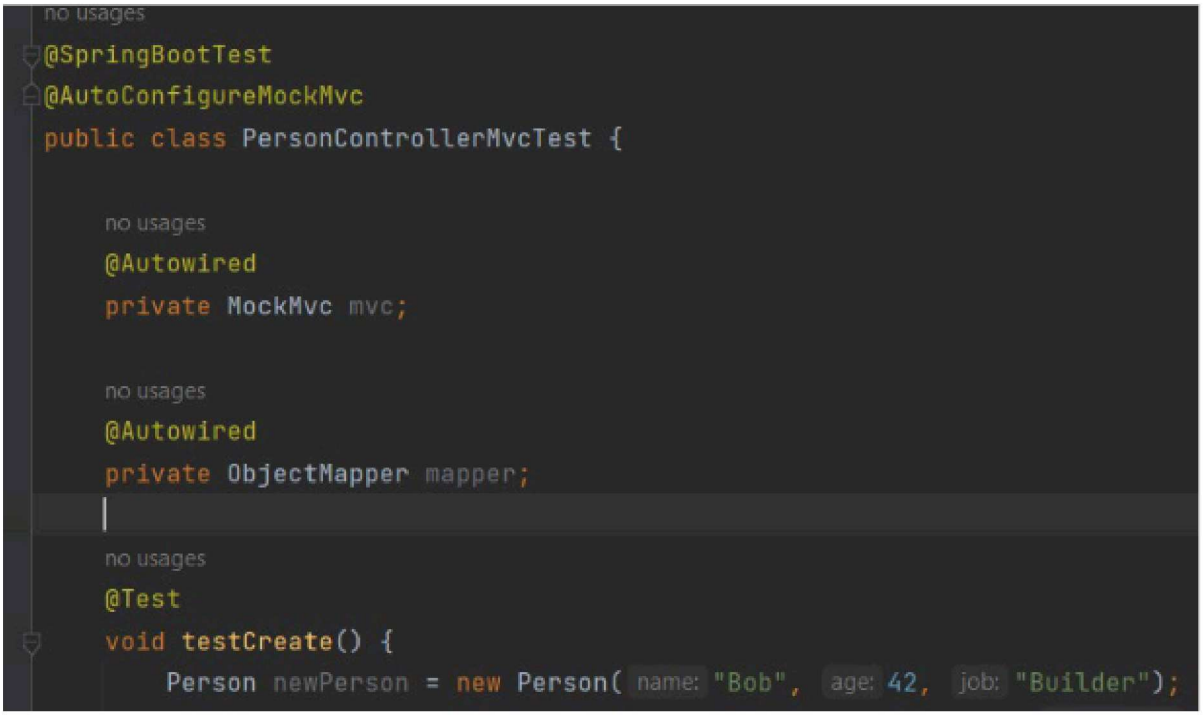
10. The **createPerson** method is set up to expect JSON data, so we need to set the content-type to application/json

```
@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.

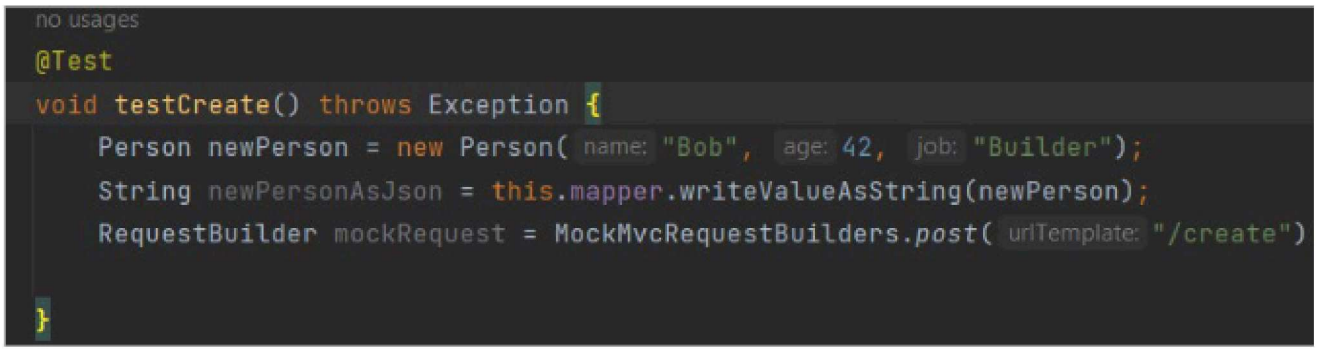
```
no usages
@Test
void testCreate() {
    Person newPerson = new Person( name: "Bob",  age: 42,  job: "Builder");
    RequestBuilder mockRequest = MockMvcRequestBuilders.post( urlTemplate: "/

}
```

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( uriTemplate: "/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( uriTemplate: "/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);
}
```

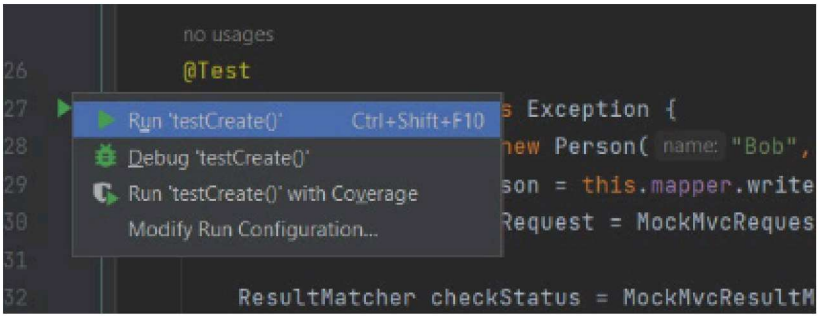
19. Send the mock request using the **perform** method from our **MockMvc** object.

```
@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: