Spring Test Framework

Spring MockMVC

Testing Spring controllers can be interesting to test. Before MockMVC existed, the options were limited to:

- Instantiating a copy of the controller class, injected the dependencies (possibly with mocks) and calling the methods by hand.
- Firing up the container and making HTTP calls by hand
- Using something like selenium to automated the HTTP calls.

There are a number of disadvantages to each of these approaches. Checking the resulting HTML can require a lot of code and can be quite brittle when the UI changes. And while the direct controller approach is fairly lightweight, it doesn't test request mapping, or any of the automatic variable conversion code. This is where MockMVC comes in, and allows testing of a large part of the MVC framework using a fluent API.

One important thing to note is that MockMVC only supports testing the Model and Controller parts of MVC. It does not test the view transformation.

Use **Spring MockMVC** to perform **integration testing** of Spring webmvc controllers. **MockMVC** class is part of Spring MVC test framework which helps in testing the controllers explicitly starting a Servlet container.

In this MockMVC tutorial, we will use it along with Spring boot's **WebMvcTest** class to execute Junit testcases which tests REST controller methods

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

- `SpringRunner` is an alias for the `SpringJUnit4ClassRunner`. It is a custom extension of JUnit's BlockJUnit4ClassRunner which provides functionality of the Spring TestContext Framework to standard JUnit tests by means of the TestContextManager and associated support classes and annotations.
- `@WebMvcTest` annotation is used for Spring MVC tests. It disables full auto-configuration and instead apply only configuration relevant to MVC tests.
- The `WebMvcTest` annotation auto-configure `MockMvc` instance as well.

- Using `RestController.class` as parameter, we are asking to initialize only one web controller and you need to provide remaining dependencies required using `Mock` objects.

```
@RunWith(SpringRunner.class)
@WebMvcTest(RestController.class)
public class TestRestController {

    @Autowired
    private MockMvc mvc;

    @Test
    public void getAllEmployeesAPI() throws Exception {
      mvc.perform( MockMvcRequestBuilders
          .get("/employees")
          .accept(MediaType.APPLICATION_JSON))
          .andDo(print())
          .andExpect(status().isOk())
          .andExpect(MockMvcResultMatchers.jsonPath("$.employees").exists())
          .andExpect(MockMvcResultMatchers.jsonPath("$.employees[*].employeeId").isNotEmpty());
    }

    @Test
    public void getEmployeeByIdAPI() throws Exception {
      mvc.perform( MockMvcRequestBuilders
          .get("/employees/{id}", 1)
          .accept(MediaType.APPLICATION_JSON))
          .andDo(print())
          .andExpect(status().isOk())
          .andExpect(MockMvcResultMatchers.jsonPath("$.employeeId").value(1));
    }

    @Test
    public void createEmployeeAPI() throws Exception {
      mvc.perform( MockMvcRequestBuilders
          .post("/employees")
          .content(asJsonString(new EmployeeVO(null, "firstName4", "lastName4", "email4@mail.com")))
          .contentType(MediaType.APPLICATION_JSON)
          .accept(MediaType.APPLICATION_JSON))
          .andExpect(status().isCreated())
          .andExpect(MockMvcResultMatchers.jsonPath("$.employeeId").exists());
    }

    public static String asJsonString(final Object obj) {
        try {
            return new ObjectMapper().writeValueAsString(obj);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    @Test
    public void deleteEmployeeAPI() throws Exception {
      mvc.perform( MockMvcRequestBuilders.delete("/employees/{id}", 1) )
            .andExpect(status().isAccepted());
    }
}
```