SUMMARY OF TESTING TIPS

Index

Author: Pablo Napoli

Based mainly on: https://amigoscode.teachable.com/p/software-testing

- Most used library imports

→ Static imports:

```
import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.core.api.Assertions.assertThatThrownBy;

import static org.mockito.Mockito.doThrow;

import static org.mockito.BDDMockito.given;
import static org.mockito.BDDMockito.then;
import static org.mockito.Mockito.doReturn;
import static org.mockito.Mockito.doNothing;
import static org.mockito.Mockito.verify;

import static org.mockito.Mockito.never;
import static org.mockito.ArgumentMatchers.any;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.hamcrest.CoreMatchers.is;
```

→ Non static imports:

```
import com.fasterxml.jackson.core.type.TypeReference;
import org.springframework.data.domain.Page;
```

- Testing JPA Queries

- For testing JPA queries I should put on top of the test class **@DataJpaTest** annotation.

- With the @DataJpaTest annotation, Spring Boot provides a convenient way to set up an environment with an embedded database to test our database queries against.

- Allow scanning of @Column annotations:
  ```
  @DataJpaTest(
  properties = { "spring.jpa.properties.javax.persistence.validation.mode=none" }
  )
  ```

- Mocking: @Mock and @InjectMocks

- Add on top of the test class: **@ExtendWith(MockitoExtension.class)**

- For mocking a class:
  ```
  @Mock
  private CustomerRepository customerRepository;
  ```

- To instantiate and inject the mock in the actual class:
  @InjectMocks
  private CustomerRegistrationService underTest;

- Mocking by org.mockito.BDDMockito

- **given**(customerRepository.selectCustomerByPhoneNumber(phoneNumber))
  .**willReturn**(Optional.empty());

- **given**(customerRepository.findById(customerId))
  .**willReturn**(Optional.of(**Mockito.mock**(Customer**.class**)));
  → There is no need to create an object.

- **given**(phoneNumberValidator.test(phoneNumber)).**willReturn**(true);

- Mocking by org.mockito.Mockito

- **doNothing**().**when(**getUserRepository()**)**.updateSector(
  matrixCaptor.capture(), codeCaptor.capture(),
  sectorIdCaptor.capture());

- **doReturn**(Optional.of(userUpdated)).**when(**getUserRepository()**)**
  .findByUserID(userUpdated.getMatrix(), userUpdated.getCode());

- "Then": assertion for invokation/no-invokation

- **then**(customerRepository).**should**().save(customerArgumentCaptor.capture());

- **then**(customerRepository).**shouldHaveNoInteractions**();

- **then**(customerRepository).**should**().save(customerArgumentCaptor.capture());

- **then**(customerRepository).**should**(**never**()).save(any());
  // Another option:
  // **then**(customerRepository).**should**().selectCustomerByPhoneNumber(phoneNumber);
  // **then**(customerRepository).**shouldHaveNoMoreInteractions**();

- "verify": Another way for assertions for invocation/no-invocation

- **verify**(customerRepository).save(customerArgumentCaptor.capture());
  → Produces the same result than:
  then(customerRepository).should().save(customerArgumentCaptor.capture());

→ Another examples:

→ verify simple invocation on mock:

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
verify(mockedList).size();
```

→ verify number of interactions with mock:

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
verify(mockedList, times(1)).size();
```

→ verify no interaction with the whole mock occurred:

```
List<String> mockedList = mock(MyList.class);
verifyZeroInteractions(mockedList);
```

→ verify no interaction with a specific method occurred:

```
List<String> mockedList = mock(MyList.class);
verify(mockedList, times(0)).size();
```

→ verify there are no unexpected interactions (this should fail):

```
List<String> mockedList = mock(MyList.class);
mockedList.size();
mockedList.clear();
verify(mockedList).size();
verifyNoMoreInteractions(mockedList);
```

→ For more examples: https://www.baeldung.com/mockito-verify

- Argument Captor

→ Creating the captor:

→ At instance level class:

```
@Captor
private ArgumentCaptor<Customer> customerCaptor;
```

→ At “then” block level from inside test case

```
ArgumentCaptor<Payment> paymentCaptor =
    ArgumentCaptor.forClass(Payment.class);
```

→ Using the captor (inside “then” block):

- then(paymentRepository).should().save(paymentCaptor**.capture()**);
- assertThat(paymentCaptor**.getValue()**).isEqualTo(payment);

- Checking thrown exception

→ From services (I'm also interested that methods below the thrown exception not to be invoked)

Example 1:

```
// when
// then
assertThatThrownBy(() -> underTest.registerNewCustomer(request))
                    .hasMessageContaining(String.format("phone number [%s] is taken", phoneNumber))
                    .isInstanceOf(IllegalStateException.class);

then(customerRepository).should(never()).save(any(Customer.class));
```

Example 2:

```
// when
assertThatThrownBy(() -> underTest.chargeCard(7L, request))
          .hasMessageContaining("Payment not received")
           .isInstanceOf(IllegalStateException.class);

// then
then(cardPaymentCharger).shouldHaveNoInteractions();
then(paymentRepository).should(never()).save(any(Payment.class));
```

Example 3:

```
doThrow(UsuIdCharInvalidException.class).when(usuIdProvider)
     .getUsuId(UserUtils.MATRIX, TYPE_A);
```

→ From a repository

```
// then
assertThatThrownBy(() -> underTest.save(customer))
          .hasMessageContaining(
                    "not-null property references a null or transient value :
                    amigos_code_prj01.customer.Customer.phoneNumber")
          .isInstanceOf(DataIntegrityViolationException.class);
```

- Testing the REST API

→ Inject MockMvc

```
@Autowired
private MockMvc mockMvc;
```

→ Invoking endpoint with path variables:

→ Given the endpoint from the REST layer:

```
@GetMapping(EndpointName.USER_MATRIX_CODE)
public ResponseEntity<UserResponseDto> findById(
    @PathVariable("matrix") String matrix,
    @PathVariable("code") String code){
  return new ResponseEntity<>(getUserController().findByCodeAndMatrix(
        matrix, code), HttpStatus.OK);
```

→ Given the test case:

```
@Test
public void whenFindByIdThenGetUserWithSucursalId() throws Exception {
  ...
  ResultActions result = mvc.perform(
        get(EndpointName.API_PREFIX + EndpointName.USER_MATRIX_CODE,
            UserUtils.MATRIX, UserUtils.code)
                .contentType(MediaType.APPLICATION_JSON));
 ...
}
```

→ Invoking endpoint with query parameters:

→ Given the endpoint from the REST layer:

```
@GetMapping(EndpointName.USER_MATRIX)
public ResponseEntity<Page<UserResponseDto>> searchByMatrix(
    @PathVariable("matrix") String matrix,
    @RequestParam(name = "page", defaultValue = "0",
          required = false) int page,
    @RequestParam(name = "size", defaultValue = "30",
          required = false) int size) {
  return new ResponseEntity<>(getUserController()
        .findByMatrix(matrix, PageRequest.of(page, size)),
        HttpStatus.OK);
}
```

→ Given the test case:

```
@Test
public void whenFindByMatrixAndExistsThenReturnAPageWithAnUser()
    throws Exception {
  ...

  mvc.perform(get(EndpointName.API_PREFIX + EndpointName.USER_MATRIX,
        UserUtils.MATRIX).param("page", "0").param("size", "1")
            .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
       …
```

→ Invoking endpoint with a request body (@RequestBody)

→ Given the endpoint from the REST layer:

```
@RequestMapping
public void makePayment(@RequestBody PaymentRequest paymentRequest) {
  paymentService.chargeCard(paymentRequest.getPayment().getCustomerId(),
paymentRequest);
}
```

→ Given the test case:

```
Payment payment = new Payment(...);
PaymentRequest paymentRequest = new PaymentRequest(payment);

ResultActions paymentResultActions = mockMvc.perform(post("/api/v1/payment")
 .contentType(MediaType.APPLICATION_JSON)
 .content(Objects.requireNonNull(objectToJson(paymentRequest))));

private String objectToJson(Object object) {
        try {
                return new ObjectMapper().writeValueAsString(object);
        } catch(JsonProcessingException e) {
                fail("Failed to convert object to json");
                return null;
        }
}
```

→ In case request's properties return an Optional:

```
<dependency>
   <groupId>com.fasterxml.jackson.datatype</groupId>
   <artifactId>jackson-datatype-jdk8</artifactId>
   <version>2.9.6</version>
</dependency>

private String objectToJson(Object object) {
        try {
                ObjectMapper mapper = new  ObjectMapper();
                mapper.registerModule(new Jdk8Module());
                return mapper.writeValueAsString(object);
        } catch(JsonProcessingException e) {
                fail("Failed to convert object to json");
                return null;
        }
}
```

→ Result handling

→ Observation: In each test case when creating an object, it'd have to be done in a utility class, but for practical purposes this is done in the same test case.

→ Result assertion by andExpect() method from the endpoint invokation

→ Given the endpoint from the REST layer:

```
@GetMapping(EndpointName.USER_MATRIX_CODE)
public ResponseEntity<UserResponseDto> findById(
      @PathVariable("matrix") String matrix,
      @PathVariable("code") String code) {
  return new ResponseEntity<>(getUserController().findByCodeAndMatrix(
          matrix, code), HttpStatus.OK);
}
```

→ Given the test case:

```
@Test
public void whenFindByIdAndExistsThenReturnAUserResponseDtoObject()
      throws Exception {
  User user = new User();
  user.setUserId(new UserPK());
  user.getId().setCode("CODE123");
  user.getId().setMatrix("VN1237");
  user.setSectorIn("S");
  user.setActive("S");
  user.setOutputFormat("J");

  doReturn(Optional.of(user)).when(getUserRepository())
          .findByUserID("VN1237", "CODE123");

  mvc.perform(get(EndpointName.API_PREFIX + EndpointName.USER_MATRIX_CODE,
          UserUtils.MATRIX, UserUtils.code)
                  .contentType(MediaType.APPLICATION_JSON))
          .andExpect(status().isOk())
          .andExpect(jsonPath("$.active", is("S")))
          .andExpect(jsonPath("$.matrix", is("VN1237")))
          .andExpect(jsonPath("$.outputFormat", is("J")))
          .andExpect(jsonPath("$.code", is("CODE123")));
}
```

→ jsonPath() accesses the JSON result by passing as the first parameter the property name as in the getter method is shown without the "get" part. The second parameter is the expected value.

→ Result assertion by checking thrown exception from ResultActions

→ Given the endpoint from the REST layer:

```
@GetMapping(EndpointName.USER_MATRIX_CODE)
public ResponseEntity<UserResponseDto> findById(
      @PathVariable("matrix") String matrix,
      @PathVariable("code") String code) {
  return new ResponseEntity<>(getUserController().findByCodeAndMatrix(
          matrix, code), HttpStatus.OK);
}
```

→ Given the test case:

```
@Test
public void whenFindByIdAndNotExistsThenResponse4xx() throws Exception {
  mvc.perform(get(EndpointName.API_PREFIX + EndpointName.USER_MATRIX_CODE,
        UserUtils.MATRIX, UserUtils.code).param("page", "0")
              .param("size", "3")
              .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().is4xxClientError())
        .andExpect(content().string("Resource not available"));
}
```

→ From UserService

```
public IUser findByMatrixAndCode(String matrix, String code)
      throws NotFoundException {
  return userRepository.findByUserID(matrix, code)
        .orElseThrow(NotFoundException::new);
}
```

→ Result assertion by extracting an object from ResultActions

→ Given the endpoint from the REST layer:

```
@GetMapping(EndpointName.USER_MATRIX_CODE)
public ResponseEntity<UserResponseDto> findById(
      @PathVariable("matrix") String matrix,
      @PathVariable("code") String code) {
  return new ResponseEntity<>(getUserController().findByCodeAndMatrix(
        matrix, code), HttpStatus.OK);
}
```

→ Given the test case:

```
@Test
public void whenFindByIdAndExistsThenReturnAUserResponseDtoObject()
      throws Exception {
  // given
  User user = new User();
  user.setUserId(new UserPK());
  user.getId().setCode("CODE123");
  user.getId().setMatrix("VN1237");
  user.setSectorIn("S");
  user.setActive("S");
  user.setOutputFormat("J");

  doReturn(Optional.of(user)).when(getUserRepository())
        .findByUserID("VN1237", "CODE123");

 // when
  ResultActions resultActions = mvc
        .perform(get(
              EndpointName.API_PREFIX + EndpointName.USER_MATRIX_CODE,
              UserUtils.MATRIX, UserUtils.code)
```

```
                .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk());

  UserResponseDto response =
        this.objectify(resultActions, UserResponseDto.class);

  // then
  assertThat(response .getActive()).isEqualTo("S");
  assertThat(response .getMatrix()).isEqualTo("VN1237");
  assertThat(response .getCode()).isEqualTo("CODE123");
  assertThat(response .getOutputFormat()).isEqualTo("J");
}
@ObjectMapper mapper;

private <T> T objectify(ResultActions resultActions, Class<T> clazz)
     throws JsonMappingException,
                JsonProcessingException,
     UnsupportedEncodingException {

  return mapper.readValue(
                resultActions.andReturn().getResponse().getContentAsString(),
                clazz);
}
```

This is the best way to be done, in which I use the andExpect() method just to check the status of the ResultActions and I could have added more checks if needed. Then, all the referred with the object obtained from the ResultActions is cheked by assertions.

→ Result assertion by extracting a Page from ResultActions

→ Given the endpoint from the REST layer:

```
@GetMapping("/customers")
public ResponseEntity<Page<CustomerResponseDto>> findAllCustomers(
                @RequestParam(name="page", defaultValue="0", required=false) int page,
                @RequestParam(name="size", defaultValue="30", required=false) int size) {
          return new ResponseEntity<>(
                        customerService.findAllCustomers(
                                      PageRequest.of(page, size))
                        .map(response -> mapper.map(response, CustomerResponseDto.class)),
                        HttpStatus.OK);
}
```

→ Given the test case:

```
@Test
public void itShouldReturnFivePageableCustomersInOnePageWhenAskingByAllCustomers()
     throws Exception {
  // given
  List<CustomerResponseDto> expectedResponse =
        IntStream.rangeClosed(1, 5).mapToObj(i -> {
          final String name = "customer_" + i;
          final String phoneNumberDigit = String.valueOf(i);
          final String phone = phoneNumberDigit + phoneNumberDigit + phoneNumberDigit +
phoneNumberDigit;
```

```
            return customerResponseDtoOf(name, phone);
        }).collect(Collectors.toList());

  // method execution
  ResultActions resultActions = mvc.perform(get(ENDPOINT)
        .param("page", "0")
        .param("size", "10")
        .contentType(MediaType.APPLICATION_JSON))
          .andExpect(jsonPath("$.numberOfElements", is(5)))
          .andExpect(jsonPath("$.totalElements", is(5)))
          .andExpect(jsonPath("$.last", is(true)))
          .andExpect(status().isOk());

  // assertion
  isExptectedContentEqualsToPageResults(expectedResponse, resultActions);
}
```

→ I should always create the following methods to extract a Page from a ResultActions and compare the result against the expected content.

→ Implement **isExptectedContentEqualsToPageResults** method:

```
private void isExptectedContentEqualsToPageResults(
    List<CustomerResponseDto> expectedContentPage,
    ResultActions resultActions) throws JsonMappingException,
    JsonProcessingException, UnsupportedEncodingException {

  List<CustomerResponseDto> returnedContentPage =
        getContentFromResultActions(resultActions);

  assertThat(expectedContentPage.size()).isEqualTo(returnedContentPage.size());

  for (int i = 0; i < returnedContentPage.size(); i++) {
    checkSimpleResult(expectedContentPage.get(i),
          returnedContentPage.get(i));
  }
}
```

→ Implement **getContentFromResultActions** method:

```
private List<CustomerResponseDto> getContentFromResultActions(ResultActions resultActions)
                    throws JsonMappingException, JsonProcessingException, UnsupportedEncodingException
{

          String responseAsString = resultActions.andReturn().getResponse().getContentAsString();

          Page<CustomerResponseDto> pageFromResult = objectMapper.readValue(responseAsString,
                              new TypeReference<RestResponsePage<CustomerResponseDto>>() {});

          return pageFromResult.getContent();
}
```

→ Implement **checkSimpleResult** method:

```
private void checkSimpleResult(CustomerResponseDto expectedDto,
          CustomerResponseDto resultDto) {
  assertThat(expectedDto.getName()).isEqualTo(resultDto.getName());
```

```
  assertThat(expectedDto.getPhoneNumber()).isEqualTo(resultDto.getPhoneNumber());
}
```

→ Create the class: **RestResponsePage**

```
import java.util.ArrayList;
import java.util.List;
import org.springframework.data.domain.PageImpl;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.databind.JsonNode;

@JsonIgnoreProperties(ignoreUnknown = true)
public class RestResponsePage<T> extends PageImpl<T> {
    private static final long serialVersionUID = 1L;

    @JsonCreator(mode = JsonCreator.Mode.PROPERTIES)
    public RestResponsePage(
                    @JsonProperty("content") List<T> content,
            @JsonProperty("number") int number, @JsonProperty("size") int size,
            @JsonProperty("totalElements") Long totalElements,
            @JsonProperty("pageable") JsonNode pageable,
            @JsonProperty("last") boolean last,
            @JsonProperty("totalPages") int totalPages,
            @JsonProperty("sort") JsonNode sort,
            @JsonProperty("first") boolean first,
            @JsonProperty("numberOfElements") int numberOfElements) {

        super(content, PageRequest.of(number, size), totalElements);
    }

    public RestResponsePage(List<T> content, Pageable pageable, long total) {
        super(content, pageable, total);
    }

    public RestResponsePage(List<T> content) {
        super(content);
    }

    public RestResponsePage() {
        super(new ArrayList<>());
    }
}
```