Testowanie w Spring

v3.0

Wprowadzenie do testowania

Testowanie

Zapoznaliśmy się z frameworkiem do testowania JUnit, kolejnym krokiem jest przetestowanie naszej aplikacji opartej o Spring framework.

Jednym z głównych założeń Springa była możliwość łatwego przetestowania aplikacji.

Dzięki wykorzystywaniu obiektów zgodnych z koncepcją **POJO** umożliwia testowanie jednostkowe.

Stosowania wstrzykiwania zależności również wspomaga testowanie jednostkowe.

Obszerny opis testowania poszczególnych elementów aplikacji opartych o **Spring** znajdziemy w dokumentacji:

https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html

Maven

Dodanie wsparcia dla testowania w naszym projekcie sprowadza się do uzupełnienia pliku **pom.xml** o następujący starter:

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

Możemy go również wskazać podczas tworzenia projektu **Spring Boot** przy użyciu **Initializer** lub możliwości naszego **IDE**.

Adnotacje

Omawiając tworzenie testu w Springu skorzystamy z udostępnionych przez framework adnotacji:

@RunWith(SpringRunner.class) - dzięki niej utworzony test będzie beanem więc będziemy mogli wstrzykiwać zależności

@ContextConfiguration - możemy wykorzystać określone konfiguracje.

@SpringBootTest - wskazuje że korzystamy z podstawowej konfiguracji dla naszej aplikacji.

@DataJpaTest - Domyślna konfiguracja dla testów JPA.

Maven

Uzupełnimy plik **pom.xml** o zależność, która umożliwi nam skorzystanie z bazy w **H2** przechowywanej w pamięci :

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

Dzięki temu nie będziemy musieli wykonywać dodatkowej konfiguracji źródła danych dla testów.

Testowanie repozytorium

Jako przypadek testowy wykorzystamy prostą encję:

```
@Entity
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String firstName;
    private String lastName;

    //gettery i settery

}
```

Testowanie repozytorium

Utworzymy repozytorium oraz 2 metody, która posłużą nam do testów:

```
public interface StudentRepository extends JpaRepository<Student, Long> {

    Student findOneByFirstName(String firstName);

    @Query("select s from Student s where s.firstName like ?1%")
    List<Student> findBySome(String some);
}
```

Testowanie repozytorium

Utworzymy repozytorium oraz 2 metody, która posłużą nam do testów:

```
public interface StudentRepository extends JpaRepository<Student, Long> {

    Student findOneByFirstName(String firstName);

    @Query("select s from Student s where s.firstName like ?1%")
    List<Student> findBySome(String some);
}
```

Metoda wyszukująca jednego Studenta po imieniu.

Testowanie repozytorium

Utworzymy repozytorium oraz 2 metody, która posłużą nam do testów:

```
public interface StudentRepository extends JpaRepository<Student, Long> {

    Student findOneByFirstName(String firstName);

    @Query("select s from Student s where s.firstName like ?1%")
    List<Student> findBySome(String some);
}
```

Metoda wyszukująca jednego Studenta po imieniu.

Metoda wyszukująca listę studentów których imię zaczyna się od podanego ciągu znaków.

Testowanie repozytorium

Następnie tworzymy klasę testu:

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class StudentRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private StudentRepository studentRepository;
}
```

Testowanie repozytorium

Następnie tworzymy klasę testu:

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class StudentRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private StudentRepository studentRepository;
}
```

Włączmy dodatkowe wsparcie dla testowania aplikacji opartej na **Spring**

Testowanie repozytorium

Następnie tworzymy klasę testu:

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class StudentRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private StudentRepository studentRepository;
}
```

Adnotacja ta ustawia standardową konfigurację dla testów - konfiguruje źródło danych - bazę **H2** - w pamięci, dokonuje konfiguracji Hibernate, włącza skanowanie encji oraz logowanie SQL.

Testowanie repozytorium

Następnie tworzymy klasę testu:

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class StudentRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private StudentRepository studentRepository;
}
```

Wstrzykujemy obiekt klasy **TestEntityManager**, jest to obiekt analogiczny do znanego już nam **EntityManager** zawierający dodatkowe metody przydatne w tworzeniu danych do testowania.

Testowanie repozytorium

Następnie tworzymy klasę testu:

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class StudentRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private StudentRepository studentRepository;
}
```

Wstrzykujemy repozytorium, którego metody będziemy testować.

Testowanie repozytorium

Dodajemy metodę testującą:

```
@Test
public void find_by_first_name_then_return_student(){
    //given
    Student john = new Student();
    john.setFirstName("John");
    entityManager.persist(john);
    //when
    Student result = studentRepository.findOneByFirstName("John");
    //then
    assertEquals(result.getFirstName(), john.getFirstName());
}
```

Testowanie repozytorium

Dodajemy metodę testującą:

```
@Test
public void find_by_first_name_then_return_student(){
    //given
    Student john = new Student();
    john.setFirstName("John");
    entityManager.persist(john);
    //when
    Student result = studentRepository.findOneByFirstName("John");
    //then
    assertEquals(result.getFirstName(), john.getFirstName());
}
```

Tworzymy i zapisujemy obiekt do źródła danych.

Testowanie repozytorium

Dodajemy metodę testującą:

```
@Test
public void find_by_first_name_then_return_student(){
    //given
    Student john = new Student();
    john.setFirstName("John");
    entityManager.persist(john);
    //when
    Student result = studentRepository.findOneByFirstName("John");
    //then
    assertEquals(result.getFirstName(), john.getFirstName());
}
```

Wywołujemy metodę repozytorium.

Testowanie repozytorium

Dodajemy metodę testującą:

```
@Test
public void find_by_first_name_then_return_student(){
    //given
    Student john = new Student();
    john.setFirstName("John");
    entityManager.persist(john);
    //when
    Student result = studentRepository.findOneByFirstName("John");
    //then
    assertEquals(result.getFirstName(), john.getFirstName());
}
```

Sprawdzamy poprawność wyniku

Testowanie repozytorium

Kolejna metoda testująca tym razem oczekujemy brak wyniku:

```
@Test
public void given_mark_then_find_john_should_not_be_null(){
    //given
    Student mark = new Student();
    mark.setFirstName("Mark");
    entityManager.persist(mark);

    //when
    Student result = studentRepository.findOneByFirstName("John");

    //then
    assertNull(result);
}
```

Testowanie repozytorium

Uzupełnimy również test dla drugiej metody repozytorium

```
@Test
public void given_jo_and_john_then_find_jo_should_return_two_elements() {
    // given
    Student jo = entityManager.persistAndFlush(new Student("jo"));
    Student john = entityManager.persistAndFlush(new Student("john"));
    // when
    List<Student> result = studentRepository.findBySome("jo");
    // then
    assertThat(result).containsExactly(jo, john);
}
```

Testowanie repozytorium

Uzupełnimy również test dla drugiej metody repozytorium

```
@Test
public void given_jo_and_john_then_find_jo_should_return_two_elements() {
    // given
    Student jo = entityManager.persistAndFlush(new Student("jo"));
    Student john = entityManager.persistAndFlush(new Student("john"));
    // when
    List<Student> result = studentRepository.findBySome("jo");
    // then
    assertThat(result).containsExactly(jo, john);
}
```

Zwróćmy uwagę że asercja jest zaimportowana z biblioteki **AssertJ** przy użyciu instrukcji importu:
`import static org.assertj.core.api.Assertions.assertThat;`

Wycofanie zmian

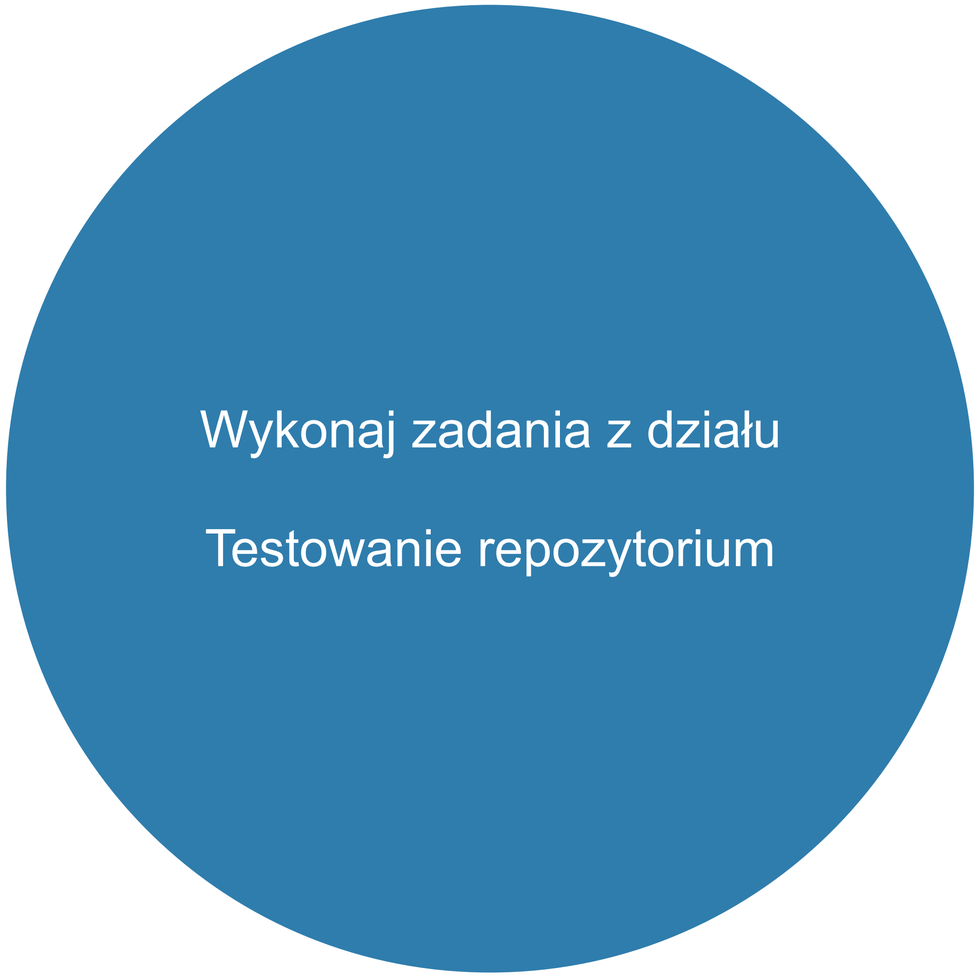
Domyślnie wszelkie zmiany wykonywane w ramach testu na bazie danych są po jego zakończeniu wycofywane.

Przy pomocy dodatkowych adnotacji mamy możliwość określenia czy zmiany powinny być wycofywane.

Więcej na ten temat znajdziemy w dokumentacji:

https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#testcontext-tx-rollback-and-commit-behavior

Zadania



Testowanie serwisu

Tworzymy interfejs

Jako przypadek testowy wykorzystamy serwis korzystający z utworzonego wcześniej repozytorium:

Najpierw utworzymy interfejs:

```
public interface StudentService {

    List<Student> listAllStudents();

    Student findByFirstName(String firstName);

    Student addStudent(Student student);

}
```

Tworzymy implementację

Tworzymy prostą implementację:

```
@Service
public class StudentServiceImpl implements StudentService {

    private StudentRepository repository;

    public StudentServiceImpl(StudentRepository repository) {
        this.repository = repository;
    }

}
```

Tworzymy implementację

Tworzymy prostą implementację:

```
@Service
public class StudentServiceImpl implements StudentService {

    private StudentRepository repository;

    public StudentServiceImpl(StudentRepository repository) {
        this.repository = repository;
    }

}
```

Oznaczamy klasę jako serwis Springa.

Tworzymy implementację

Tworzymy prostą implementację:

```
@Service
public class StudentServiceImpl implements StudentService {

    private StudentRepository repository;

    public StudentServiceImpl(StudentRepository repository) {
        this.repository = repository;
    }

}
```

Wstrzykujemy za pomocą konstruktora repozytorium.

Tworzymy implementację

Uzupełniamy implementację metod:

```
@Override
public List<Student> listAllStudents() {
    return repository.findAll();
}

@Override
public Student findByFirstName(String firstName) {
    return repository.findOneByFirstName(firstName);
}

@Override
public Student addStudent(Student student) {
    return repository.save(student);
}
```

Tworzymy klasę testową

```
public class StudentServiceTest {

    private static final Logger log =
                    LoggerFactory.getLogger(StudentServiceTest.class);

    private StudentService service;
    private StudentRepository repository;
    @Before
    public void setUp() {
        repository = mock(StudentRepository.class);
        service = new StudentServiceImpl(repository);
    }
}
```

Tworzymy klasę testową

```
public class StudentServiceTest {

    private static final Logger log =
                        LoggerFactory.getLogger(StudentServiceTest.class);

    private StudentService service;
    private StudentRepository repository;
    @Before
    public void setUp() {
        repository = mock(StudentRepository.class);
        service = new StudentServiceImpl(repository);
    }
}
```

Tworzymy logger - przyda nam się przy początkowej pracy z testami.

Tworzymy klasę testową

```
public class StudentServiceTest {

    private static final Logger log =
                    LoggerFactory.getLogger(StudentServiceTest.class);

    private StudentService service;
    private StudentRepository repository;
    @Before
    public void setUp() {
        repository = mock(StudentRepository.class);
        service = new StudentServiceImpl(repository);
    }
}
```

Wykorzystując statyczną metodę **`org.mockito.Mockito.mock`** tworzymy imitację wymaganego przez serwis obiektu repozytorium.

Tworzymy klasę testową

```
public class StudentServiceTest {

    private static final Logger log =
                    LoggerFactory.getLogger(StudentServiceTest.class);

    private StudentService service;
    private StudentRepository repository;
    @Before
    public void setUp() {
        repository = mock(StudentRepository.class);
        service = new StudentServiceImpl(repository);
    }
}
```

Tworzymy obiekt serwisu podając wymaganą przez niego zależność w konstruktorze klasy.

Tworzymy test

```
@Test
public void when_searching_john_then_return_object() {
    // given
    Student john = new Student("John");
    when(repository.findOneByFirstName("John")).thenReturn(john);
    // when
    Student student = service.findByFirstName("John");
    // then
    assertEquals(student.getFirstName(), "John");
}
```

Tworzymy test

```
@Test
public void when_searching_john_then_return_object() {
    // given
    Student john = new Student("John");
    when(repository.findOneByFirstName("John")).thenReturn(john);
    // when
    Student student = service.findByFirstName("John");
    // then
    assertEquals(student.getFirstName(), "John");
}
```

Tworzymy obiekt którym będziemy się posługiwać podczas testów.

Tworzymy test

```
@Test
public void when_searching_john_then_return_object() {
    // given
    Student john = new Student("John");
    when(repository.findOneByFirstName("John")).thenReturn(john);
    // when
    Student student = service.findByFirstName("John");
    // then
    assertEquals(student.getFirstName(), "John");
}
```

Przy pomocy statycznej metody **org.mockito.Mockito.when** opisujemy co ma się wydarzyć w przypadku wywołania określonej metody na obiekcie repozytorium.

Tworzymy test

```
@Test
public void when_searching_john_then_return_object() {
    // given
    Student john = new Student("John");
    when(repository.findOneByFirstName("John")).thenReturn(john);
    // when
    Student student = service.findByFirstName("John");
    // then
    assertEquals(student.getFirstName(), "John");
}
```

Przy wywołaniu metody **findOneByFirstName** ma zostać zwrócony wcześniej zdefiniowany obiekt.

Tworzymy test

```
@Test
public void when_searching_john_then_return_object() {
    // given
    Student john = new Student("John");
    when(repository.findOneByFirstName("John")).thenReturn(john);
    // when
    Student student = service.findByFirstName("John");
    // then
    assertEquals(student.getFirstName(), "John");
}
```

Wywołujemy metodę serwisu.

Tworzymy test

```
@Test
public void when_searching_john_then_return_object() {
    // given
    Student john = new Student("John");
    when(repository.findOneByFirstName("John")).thenReturn(john);
    // when
    Student student = service.findByFirstName("John");
    // then
    assertEquals(student.getFirstName(), "John");
}
```

Weryfikujemy otrzymane wyniki.

Tworzymy test

```
@Test
public void when_save_student_then_it_is_returned_correctly() {
    // given
    Student student = new Student("John");
    when(repository.save(student)).thenReturn(student);
    // when
    Student result = service.addStudent(student);
    // then
    assertNotNull(result);
    assertEquals(student.getFirstName(), result.getFirstName());
}
```

Tworzymy test

```
@Test
public void when_save_student_then_it_is_returned_correctly() {
    // given
    Student student = new Student("John");
    when(repository.save(student)).thenReturn(student);
    // when
    Student result = service.addStudent(student);
    // then
    assertNotNull(result);
    assertEquals(student.getFirstName(), result.getFirstName());
}
```

Tworzymy obiekt którym będziemy się posługiwać podczas testów.

Tworzymy test

```
@Test
public void when_save_student_then_it_is_returned_correctly() {
    // given
    Student student = new Student("John");
    when(repository.save(student)).thenReturn(student);
    // when
    Student result = service.addStudent(student);
    // then
    assertNotNull(result);
    assertEquals(student.getFirstName(), result.getFirstName());
}
```

Przy pomocy statycznej metody **org.mockito.Mockito.when** opisujemy co ma się wydarzyć w przypadku wywołania określonej metody na obiekcie repozytorium.

Tworzymy test

```
@Test
public void when_save_student_then_it_is_returned_correctly() {
    // given
    Student student = new Student("John");
    when(repository.save(student)).thenReturn(student);
    // when
    Student result = service.addStudent(student);
    // then
    assertNotNull(result);
    assertEquals(student.getFirstName(), result.getFirstName());
}
```

Przy wywołaniu metody **save** ma zostać zwrócony wcześniej zdefiniowany obiekt.

Tworzymy test

```
@Test
public void when_save_student_then_it_is_returned_correctly() {
    // given
    Student student = new Student("John");
    when(repository.save(student)).thenReturn(student);
    // when
    Student result = service.addStudent(student);
    // then
    assertNotNull(result);
    assertEquals(student.getFirstName(), result.getFirstName());
}
```

Wywołujemy metodę serwisu.

Tworzymy test

```
@Test
public void when_save_student_then_it_is_returned_correctly() {
    // given
    Student student = new Student("John");
    when(repository.save(student)).thenReturn(student);
    // when
    Student result = service.addStudent(student);
    // then
    assertNotNull(result);
    assertEquals(student.getFirstName(), result.getFirstName());
}
```

Weryfikujemy otrzymane wyniki.

Tworzymy test

Zamiast tworzyć przy pomocy statycznej metody **mock** atrapę naszego repozytorium możemy skorzystać z adnotacji, dzięki temu utworzą się one automatycznie:

```
@RunWith(MockitoJUnitRunner.class)
public class StudentServiceTestAnnotation {

    private StudentService service;

    @Mock
    private StudentRepository repository;

    @Before
    public void setUp() {
        service = new StudentServiceImpl(repository);
    }
}
```

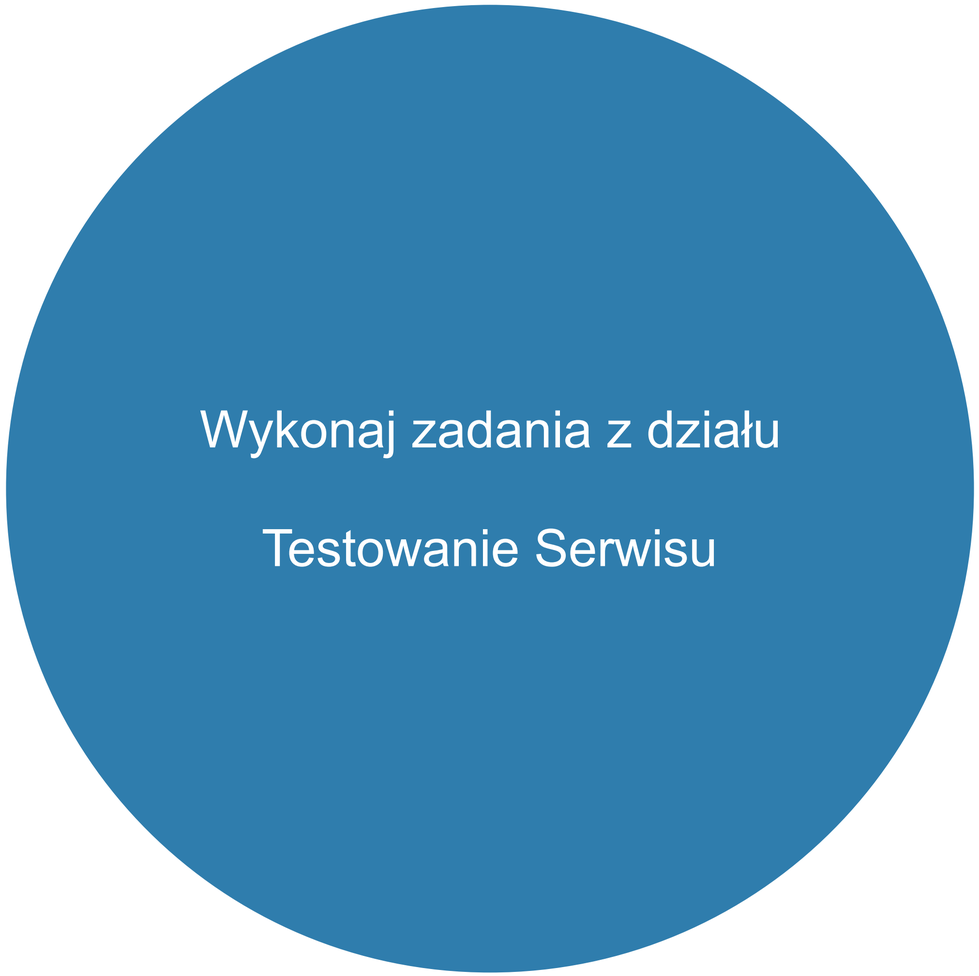
Tworzymy test

Zamiast tworzyć przy pomocy statycznej metody **mock** atrapę naszego repozytorium możemy skorzystać z adnotacji, dzięki temu utworzą się one automatycznie:

```
@RunWith(MockitoJUnitRunner.class)
public class StudentServiceTestAnnotation {

    private StudentService service;

    @Mock
    private StudentRepository repository;

    @Before
    public void setUp() {
        service = new StudentServiceImpl(repository);
    }
}
```

Wykorzystujemy adnotacje **@RunWith(MockitoJUnitRunner.class)** oraz **@Mock**.

Zadania



Testowanie kontrolera

Tworzymy kontroler

Jako przypadek testowy wykorzystamy prosty kontroler oraz zawarte w nim akcje:

```
@Controller
public class PageController {

    @GetMapping("/page")
    public String page() {
        return "page/index";
    }

    @GetMapping("/")
    public String home() {
        return "page/home";
    }

}
```

Test klasy

Klasa ta oprócz dodatkowych adnotacji jest zwykłą klasą Javy, którą możemy przetestować jednostkowo z wykorzystaniem JUnit w znany nam już sposób:

```
public class PageControllerTest {
    private static PageController pageController;
    private static String PAGE_VIEW_NAME = "page/index";

    @BeforeClass
    public static void beforeClass() {
        pageController = new PageController();
    }
    @Test
    public void test_home_action_return_page() {
        assertEquals(pageController.page(), PAGE_VIEW_NAME);
    }
}
```

Test klasy

Klasa ta oprócz dodatkowych adnotacji jest zwykłą klasą Javy, którą możemy przetestować jednostkowo z wykorzystaniem JUnit w znany nam już sposób:

```
public class PageControllerTest {
    private static PageController pageController;
    private static String PAGE_VIEW_NAME = "page/index";

    @BeforeClass
    public static void beforeClass() {
        pageController = new PageController();
    }
    @Test
    public void test_home_action_return_page() {
        assertEquals(pageController.page(), PAGE_VIEW_NAME);
    }
}
```

Tworzymy statyczne zmienne, które będziemy wykorzystywać w testach.

Test klasy

Klasa ta oprócz dodatkowych adnotacji jest zwykłą klasą Javy, którą możemy przetestować jednostkowo z wykorzystaniem JUnit w znany nam już sposób:

```
public class PageControllerTest {
    private static PageController pageController;
    private static String PAGE_VIEW_NAME = "page/index";

    @BeforeClass
    public static void beforeClass() {
        pageController = new PageController();
    }
    @Test
    public void test_home_action_return_page() {
        assertEquals(pageController.page(), PAGE_VIEW_NAME);
    }
}
```

W metodzie oznaczonej adnotacją **@BeforeClass** inicjujemy obiekt klasy **PageController**.

Test klasy

Klasa ta oprócz dodatkowych adnotacji jest zwykłą klasą Javy, którą możemy przetestować jednostkowo z wykorzystaniem JUnit w znany nam już sposób:

```
public class PageControllerTest {
    private static PageController pageController;
    private static String PAGE_VIEW_NAME = "page/index";

    @BeforeClass
    public static void beforeClass() {
        pageController = new PageController();
    }
    @Test
    public void test_home_action_return_page() {
        assertEquals(pageController.page(), PAGE_VIEW_NAME);
    }
}
```

Sprawdzamy czy zwracana przez metodę nazwa widoku jest równa oczekiwanej przez nas wartości.

Test klasy

Test ten mimo że działa prawidłowo, nie sprawdza w pełni naszego prostego kontrolera, tzn, nie wiemy czy wywołane prawidłowo żądanie **GET** da nam oczekiwany przez nas widok, czy w ramach przekazywanego do widoku modelu znajdują się odpowiednie dane.

W celu przetestowania żądania posłużymy się klasą **MockMvc** jest ona dostarczana przez **Springa**, a jej zadaniem jest symulacja akcji przeglądarki, np. wywoływanie żądania - takie wywołanie umieścimy wewnątrz metody testowej.

MockMvc

Poniżej przykład z wykorzystaniem **MockMvc** - akcja dostępna pod adresem **/page** :

```
@RunWith(SpringRunner.class)
public class PageControllerMockTestStandalone {
    private static PageController pageController;
    private static String PAGE_VIEW_NAME_PAGE = "page/index";
    @BeforeClass
    public static void setUp() {pageController = new PageController();}
    @Test
    public void test_home_action_return_index() throws Exception {
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(pageController).build();
        mockMvc.perform(MockMvcRequestBuilders.get("/page"))
                .andExpect(MockMvcResultMatchers.view().name(PAGE_VIEW_NAME_PAGE))
    }
}
```

MockMvc

Poniżej przykład z wykorzystaniem **MockMvc** - akcja dostępna pod adresem **/page** :

```
@RunWith(SpringRunner.class)
public class PageControllerMockTestStandalone {
    private static PageController pageController;
    private static String PAGE_VIEW_NAME_PAGE = "page/index";
    @BeforeClass
    public static void setUp() {pageController = new PageController();}
    @Test
    public void test_home_action_return_index() throws Exception {
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(pageController).build();
        mockMvc.perform(MockMvcRequestBuilders.get("/page"))
                .andExpect(MockMvcResultMatchers.view().name(PAGE_VIEW_NAME_PAGE))
    }
}
```

Przy pomocy statycznej metody **standaloneSetup** tworzymy instancję kontrolera w sposób programistyczny, bez fizycznego uruchomienia go w kontenerze servletów.

MockMvc

Poniżej przykład z wykorzystaniem **MockMvc** - akcja dostępna pod adresem **/page** :

```
@RunWith(SpringRunner.class)
public class PageControllerMockTestStandalone {
    private static PageController pageController;
    private static String PAGE_VIEW_NAME_PAGE = "page/index";
    @BeforeClass
    public static void setUp() {pageController = new PageController();}
    @Test
    public void test_home_action_return_index() throws Exception {
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(pageController).build();
        mockMvc.perform(MockMvcRequestBuilders.get("/page"))
                .andExpect(MockMvcResultMatchers.view().name(PAGE_VIEW_NAME_PAGE))
    }
}
```

Wywołujemy żądanie typu **GET** z url **/page**.

MockMvc

Poniżej przykład z wykorzystaniem **MockMvc** - akcja dostępna pod adresem **/page** :

```
@RunWith(SpringRunner.class)
public class PageControllerMockTestStandalone {
    private static PageController pageController;
    private static String PAGE_VIEW_NAME_PAGE = "page/index";
    @BeforeClass
    public static void setUp() {pageController = new PageController();}
    @Test
    public void test_home_action_return_index() throws Exception {
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(pageController).build();
        mockMvc.perform(MockMvcRequestBuilders.get("/page"))
                .andExpect(MockMvcResultMatchers.view().name(PAGE_VIEW_NAME_PAGE))
    }
}
```

Sprawdzamy czy nazwa widoku jest równa nazwie oczekiwanej.

MockMvc

Uzupełniamy klasę testującą o analogiczny test dla akcji **home**.

Tym razem korzystamy ze statycznych importów - podobnie jak w przypadku poznanego już **JUnit**.

```
private static String PAGE_VIEW_NAME_HOME = "page/home";
@Test
public void testHomePage() throws Exception {
    PageController controller = new PageController();
    MockMvc mockMvc = standaloneSetup(controller).build();
    mockMvc.perform(get("/")).andExpect(view().name(PAGE_VIEW_NAME_HOME));
}
```

MockMvc

Uzupełniamy klasę testującą o analogiczny test dla akcji **home**.

Tym razem korzystamy ze statycznych importów - podobnie jak w przypadku poznanego już **JUnit**.

```
private static String PAGE_VIEW_NAME_HOME = "page/home";
@Test
public void testHomePage() throws Exception {
    PageController controller = new PageController();
    MockMvc mockMvc = standaloneSetup(controller).build();
    mockMvc.perform(get("/")).andExpect(view().name(PAGE_VIEW_NAME_HOME));
}
```

Jest to odpowiednik wywołania **MockMvcBuilders.standaloneSetup** .

MockMvc

Uzupełniamy klasę testującą o analogiczny test dla akcji **home**.

Tym razem korzystamy ze statycznych importów - podobnie jak w przypadku poznanego już **JUnit**.

```
private static String PAGE_VIEW_NAME_HOME = "page/home";
@Test
public void testHomePage() throws Exception {
    PageController controller = new PageController();
    MockMvc mockMvc = standaloneSetup(controller).build();
    mockMvc.perform(get("/")).andExpect(view().name(PAGE_VIEW_NAME_HOME));
}
```

Analogicznie wywołania **MockMvcRequestBuilders.get** oraz **MockMvcResultMatchers.view** .

Importy statyczne

Korzystamy z następujących instrukcji importu:

```
import static org.springframework.test
    .web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test
    .web.servlet.result.MockMvcResultMatchers.view;
import static org.springframework
    .test.web.servlet.setup.MockMvcBuilders.standaloneSetup;
```

Rozbudowa testu

Rozbudujemy nasz test o dodatkowe możliwości:

```
@Test
public void test_home_action_return_index_static_import() throws Exception {
    MockMvc mockMvc = standaloneSetup(pageController).build();
    mockMvc.perform(get("/"))
            .andExpect(view().name(PAGE_VIEW_NAME_HOME))
            .andExpect(status().isOk())
            .andDo(print());
}
```

Rozbudowa testu

Rozbudujemy nasz test o dodatkowe możliwości:

```
@Test
public void test_home_action_return_index_static_import() throws Exception {
    MockMvc mockMvc = standaloneSetup(pageController).build();
    mockMvc.perform(get("/"))
            .andExpect(view().name(PAGE_VIEW_NAME_HOME))
            .andExpect(status().isOk())
            .andDo(print());
}
```

Sprawdzamy czy status odpowiedzi jest równy 200.

Rozbudowa testu

Rozbudujemy nasz test o dodatkowe możliwości:

```
@Test
public void test_home_action_return_index_static_import() throws Exception {
    MockMvc mockMvc = standaloneSetup(pageController).build();
    mockMvc.perform(get("/"))
            .andExpect(view().name(PAGE_VIEW_NAME_HOME))
            .andExpect(status().isOk())
            .andDo(print());
}
```

Wyświetlamy wywołanie na konsoli.

WebApplicationContext

Poprzedni test nie uruchamiał konfiguracji naszej aplikacji był więc bardziej zbliżony do testu jednostkowego.

W poniższy sposób załadujemy elementy kontekstu Springa:

```
@RunWith(SpringRunner.class)
@WebMvcTest(controllers = StudentController.class)
public class StudentControllerTest {
    private MockMvc mockMvc;
    @Autowired
    private WebApplicationContext webApplicationContext;

    @Before
    public void setUp() {
        mockMvc = webAppContextSetup(webApplicationContext).build();
    }
}
```

WebApplicationContext

Poprzedni test nie uruchamiał konfiguracji naszej aplikacji był więc bardziej zbliżony do testu jednostkowego.

W poniższy sposób załadujemy elementy kontekstu Springa:

```
@RunWith(SpringRunner.class)
@WebMvcTest(controllers = StudentController.class)
public class StudentControllerTest {
    private MockMvc mockMvc;
    @Autowired
    private WebApplicationContext webApplicationContext;

    @Before
    public void setUp() {
        mockMvc = webAppContextSetup(webApplicationContext).build();
    }
}
```

Określamy kontroler który testujemy.

WebApplicationContext

Poprzedni test nie uruchamiał konfiguracji naszej aplikacji był więc bardziej zbliżony do testu jednostkowego.

W poniższy sposób załadujemy elementy kontekstu Springa:

```
@RunWith(SpringRunner.class)
@WebMvcTest(controllers = StudentController.class)
public class StudentControllerTest {
    private MockMvc mockMvc;
    @Autowired
    private WebApplicationContext webApplicationContext;

    @Before
    public void setUp() {
        mockMvc = webAppContextSetup(webApplicationContext).build();
    }
}
```

Wstrzykujemy obiekt kontekstu.

WebApplicationContext

Poprzedni test nie uruchamiał konfiguracji naszej aplikacji był więc bardziej zbliżony do testu jednostkowego.

W poniższy sposób załadujemy elementy kontekstu Springa:

```
@RunWith(SpringRunner.class)
@WebMvcTest(controllers = StudentController.class)
public class StudentControllerTest {
    private MockMvc mockMvc;
    @Autowired
    private WebApplicationContext webApplicationContext;

    @Before
    public void setUp() {
        mockMvc = webAppContextSetup(webApplicationContext).build();
    }
}
```

Tworzymy atrapę.

WebApplicationContext

Dodajemy metodę testową:

```
private final String STUDENT_LIST_ACTION_VIEW = "student/studentList";
@Test
public void test_listAction_contains_model_list() throws Exception {
    mockMvc.perform(get("/students/list"))
        .andExpect(model().attributeExists("list"))
        .andExpect(status().isOk())
        .andExpect(content().contentType("text/html;charset=UTF-8"))
        .andExpect(content().string(containsString("Insert title")))
        .andExpect(view().name(STUDENT_LIST_ACTION_VIEW));
}
```

WebApplicationContext

Dodajemy metodę testową:

```
private final String STUDENT_LIST_ACTION_VIEW = "student/studentList";
@Test
public void test_listAction_contains_model_list() throws Exception {
    mockMvc.perform(get("/students/list"))
        .andExpect(model().attributeExists("list"))
        .andExpect(status().isOk())
        .andExpect(content().contentType("text/html;charset=UTF-8"))
        .andExpect(content().string(containsString("Insert title")))
        .andExpect(view().name(STUDENT_LIST_ACTION_VIEW));
}
```

Definiujemy stałą z nazwą widoku.

WebApplicationContext

Dodajemy metodę testową:

```
private final String STUDENT_LIST_ACTION_VIEW = "student/studentList";
@Test
public void test_listAction_contains_model_list() throws Exception {
    mockMvc.perform(get("/students/list"))
            .andExpect(model().attributeExists("list"))
            .andExpect(status().isOk())
            .andExpect(content().contentType("text/html;charset=UTF-8"))
            .andExpect(content().string(containsString("Insert title")))
            .andExpect(view().name(STUDENT_LIST_ACTION_VIEW));
}
```

Wywołujemy akcję dostępną pod adresem **/students/list** .

WebApplicationContext

Dodajemy metodę testową:

```
private final String STUDENT_LIST_ACTION_VIEW = "student/studentList";
@Test
public void test_listAction_contains_model_list() throws Exception {
    mockMvc.perform(get("/students/list"))
        .andExpect(model().attributeExists("list"))
        .andExpect(status().isOk())
        .andExpect(content().contentType("text/html;charset=UTF-8"))
        .andExpect(content().string(containsString("Insert title")))
        .andExpect(view().name(STUDENT_LIST_ACTION_VIEW));
}
```

Sprawdzamy, czy model zawiera atrybut o nazwie **list** .

WebApplicationContext

Dodajemy metodę testową:

```
private final String STUDENT_LIST_ACTION_VIEW = "student/studentList";
@Test
public void test_listAction_contains_model_list() throws Exception {
    mockMvc.perform(get("/students/list"))
        .andExpect(model().attributeExists("list"))
        .andExpect(status().isOk())
        .andExpect(content().contentType("text/html;charset=UTF-8"))
        .andExpect(content().string(containsString("Insert title")))
        .andExpect(view().name(STUDENT_LIST_ACTION_VIEW));
}
```

Weryfikujemy poprawny status odpowiedzi.

WebApplicationContext

Dodajemy metodę testową:

```
private final String STUDENT_LIST_ACTION_VIEW = "student/studentList";
@Test
public void test_listAction_contains_model_list() throws Exception {
    mockMvc.perform(get("/students/list"))
        .andExpect(model().attributeExists("list"))
        .andExpect(status().isOk())
        .andExpect(content().contentType("text/html;charset=UTF-8"))
        .andExpect(content().string(containsString("Insert title")))
        .andExpect(view().name(STUDENT_LIST_ACTION_VIEW));
}
```

Sprawdzamy typ zwracanej wartości.

WebApplicationContext

Dodajemy metodę testową:

```
private final String STUDENT_LIST_ACTION_VIEW = "student/studentList";
@Test
public void test_listAction_contains_model_list() throws Exception {
    mockMvc.perform(get("/students/list"))
        .andExpect(model().attributeExists("list"))
        .andExpect(status().isOk())
        .andExpect(content().contentType("text/html;charset=UTF-8"))
        .andExpect(content().string(containsString("Insert title")))
        .andExpect(view().name(STUDENT_LIST_ACTION_VIEW));
}
```

Sprawdzamy, czy zwracana zawartość zawiera zdefiniowany napis.

WebApplicationContext

Dodajemy metodę testową:

```
private final String STUDENT_LIST_ACTION_VIEW = "student/studentList";
@Test
public void test_listAction_contains_model_list() throws Exception {
    mockMvc.perform(get("/students/list"))
        .andExpect(model().attributeExists("list"))
        .andExpect(status().isOk())
        .andExpect(content().contentType("text/html;charset=UTF-8"))
        .andExpect(content().string(containsString("Insert title")))
        .andExpect(view().name(STUDENT_LIST_ACTION_VIEW));
}
```

Sprawdzamy, czy widok jest równy oczekiwanemu.

Kontroler

Dodajemy prostą akcję spełniającą nasz test:

```
@Controller
public class StudentController {

    @RequestMapping("/students/list")
    public String list(Model model) {
        model.addAttribute("list", new ArrayList());
        return "student/studentList";
    }
}
```

Mockowanie zależności

Nasza akcja zamiast tworzenia nowej listy, skorzysta z serwisu, który za pomocą repozytorium taką listę nam dostarczy.

```
@Controller
public class StudentController {
    private StudentService service;

    public StudentController(StudentService service) {
        this.service = service;
    }

    @RequestMapping("/students/list")
    public String list(Model model) {
        model.addAttribute("list", service.listAllStudents());
        return "student/studentList";
    }
}
```

Mockowanie zależności

Modyfikujemy konfigurację testu.

```
@MockBean
private StudentService studentService;

@Before
public void setUp() {
    mockMvc = webAppContextSetup(webApplicationContext).build();

    List<Student> students = Arrays.asList(new Student("Jan"),
                    new Student("Janek"), new Student("Janusz"));
    when(this.studentService.listAllStudents()).thenReturn(students);
}
```

Mockowanie zależności

Modyfikujemy konfigurację testu.

```
@MockBean
private StudentService studentService;

@Before
public void setUp() {
    mockMvc = webAppContextSetup(webApplicationContext).build();

    List<Student> students = Arrays.asList(new Student("Jan"),
                    new Student("Janek"), new Student("Janusz"));
    when(this.studentService.listAllStudents()).thenReturn(students);
}
```

Tworzymy mock dla naszego serwisu.

Mockowanie zależności

Modyfikujemy konfigurację testu.

```
@MockBean
private StudentService studentService;

@Before
public void setUp() {
    mockMvc = webAppContextSetup(webApplicationContext).build();

    List<Student> students = Arrays.asList(new Student("Jan"),
                    new Student("Janek"), new Student("Janusz"));
    when(this.studentService.listAllStudents()).thenReturn(students);
}
```

Tworzymy listę która zostanie zwrócona przy wywołaniu metody serwisu.

Mockowanie zależności

Modyfikujemy konfigurację testu.

```
@MockBean
private StudentService studentService;

@Before
public void setUp() {
    mockMvc = webAppContextSetup(webApplicationContext).build();

    List<Student> students = Arrays.asList(new Student("Jan"),
                    new Student("Janek"), new Student("Janusz"));
    when(this.studentService.listAllStudents()).thenReturn(students);
}
```

Określamy zachowanie w przypadku wywołania metody **listAllStudents** serwisu **studentService** .

Mockowanie zależności

Modyfikujemy test.

```
@Test
public void test_listAction_contains_model_list() throws Exception {
    assertThat(this.studentService).isNotNull();

    mockMvc.perform(get("/students/list"))
            .andExpect(model().attributeExists("list"))
            .andExpect(model().attribute("list", hasSize(3)))
            .andExpect(model().attribute("list",
                    hasItem(anyOf(hasProperty("firstName"), is("Jan")))))
            .andExpect(status().isOk())
            .andExpect(content().contentType("text/html;charset=UTF-8"))
            .andExpect(content().string(containsString("Janek")))
            .andExpect(view().name(STUDENT_LIST_ACTION_VIEW))
            .andDo(print());
}
```

Mockowanie zależności

Modyfikujemy test.

```
@Test
public void test_listAction_contains_model_list() throws Exception {
    assertThat(this.studentService).isNotNull();

    mockMvc.perform(get("/students/list"))
            .andExpect(model().attributeExists("list"))
            .andExpect(model().attribute("list", hasSize(3)))
            .andExpect(model().attribute("list",
                    hasItem(anyOf(hasProperty("firstName"), is("Jan")))))
            .andExpect(status().isOk())
            .andExpect(content().contentType("text/html;charset=UTF-8"))
            .andExpect(content().string(containsString("Janek")))
            .andExpect(view().name(STUDENT_LIST_ACTION_VIEW))
            .andDo(print());
}
```

Mockowanie zależności

Modyfikujemy test.

```
@Test
public void test_listAction_contains_model_list() throws Exception {
    assertThat(this.studentService).isNotNull();

    mockMvc.perform(get("/students/list"))
            .andExpect(model().attributeExists("list"))
            .andExpect(model().attribute("list", hasSize(3)))
            .andExpect(model().attribute("list",
                    hasItem(anyOf(hasProperty("firstName"), is("Jan")))))
            .andExpect(status().isOk())
            .andExpect(content().contentType("text/html;charset=UTF-8"))
            .andExpect(content().string(containsString("Janek")))
            .andExpect(view().name(STUDENT_LIST_ACTION_VIEW))
            .andDo(print());
}
```

Mockowanie zależności

Modyfikujemy test.

```
@Test
public void test_listAction_contains_model_list() throws Exception {
    assertThat(this.studentService).isNotNull();

    mockMvc.perform(get("/students/list"))
            .andExpect(model().attributeExists("list"))
            .andExpect(model().attribute("list", hasSize(3)))
            .andExpect(model().attribute("list",
                    hasItem(anyOf(hasProperty("firstName"), is("Jan")))))
            .andExpect(status().isOk())
            .andExpect(content().contentType("text/html;charset=UTF-8"))
            .andExpect(content().string(containsString("Janek")))
            .andExpect(view().name(STUDENT_LIST_ACTION_VIEW))
            .andDo(print());
}
```

Mockowanie zależności

Modyfikujemy test.

```
@Test
public void test_listAction_contains_model_list() throws Exception {
    assertThat(this.studentService).isNotNull();

    mockMvc.perform(get("/students/list"))
            .andExpect(model().attributeExists("list"))
            .andExpect(model().attribute("list", hasSize(3)))
            .andExpect(model().attribute("list",
                    hasItem(anyOf(hasProperty("firstName"), is("Jan")))))
            .andExpect(status().isOk())
            .andExpect(content().contentType("text/html;charset=UTF-8"))
            .andExpect(content().string(containsString("Janek")))
            .andExpect(view().name(STUDENT_LIST_ACTION_VIEW))
            .andDo(print());
}
```

Ustawianie parametrów

Jako przykład utworzymy akcję, która w adresie url otrzyma identyfikator a następnie pobierze i przekaże do modelu obiekt klasy **Student** .

```
@GetMapping("/students/show/{id}")
public String show(Model model, @PathVariable long id) {
    Student student = service.getStudentById(id);
    model.addAttribute("student", student);
    return "student/showStudent";
}
```

Ustawianie parametrów

Uzupełniamy metodę **`setUp()`** o następującą definicję

```
Student kowalski = new Student("Jan");
kowalski.setLastName("Kowalski");
when(this.studentService.getStudentById(1L)).thenReturn(kowalski);
```

Ustawianie parametrów

Uzupełniamy metodę **`setUp()`** o następującą definicję

```
Student kowalski = new Student("Jan");
kowalski.setLastName("Kowalski");
when(this.studentService.getStudentById(1L)).thenReturn(kowalski);
```

Tworzymy nowy obiekt.

Ustawianie parametrów

Uzupełniamy metodę **`setUp()`** o następującą definicję

```
Student kowalski = new Student("Jan");
kowalski.setLastName("Kowalski");
when(this.studentService.getStudentById(1L)).thenReturn(kowalski);
```

Tworzymy nowy obiekt.

Ustawiamy wartość atrybutu.

Ustawianie parametrów

Uzupełniamy metodę **`setUp()`** o następującą definicję

```
Student kowalski = new Student("Jan");
kowalski.setLastName("Kowalski");
when(this.studentService.getStudentById(1L)).thenReturn(kowalski);
```

Tworzymy nowy obiekt.

Ustawiamy wartość atrybutu.

Określamy zachowanie w przypadku wywołania metody serwisu: **getStudentById**.

Ustawianie parametrów

Test dla powyższej akcji będzie wyglądał następująco:

```
@Test
public void test_show_student() throws Exception {
    mockMvc.perform(get("/students/show/{id}", 1L))
        .andExpect(status().isOk())
        .andExpect(view().name("student/showStudent"))
        .andExpect(model().attributeExists("student"))
        .andExpect(model().attribute("student",
                    hasProperty("lastName", is("Kowalski"))))
    ;
}
```

Ustawianie parametrów

Test dla powyższej akcji będzie wyglądał następująco:

```
@Test
public void test_show_student() throws Exception {
    mockMvc.perform(get("/students/show/{id}", 1L))
            .andExpect(status().isOk())
            .andExpect(view().name("student/showStudent"))
            .andExpect(model().attributeExists("student"))
            .andExpect(model().attribute("student",
                    hasProperty("lastName", is("Kowalski"))))
    ;
}
```

Wywołujemy akcję, określając jej parametr adresu **1L** .

Ustawianie parametrów

Test dla powyższej akcji będzie wyglądał następująco:

```
@Test
public void test_show_student() throws Exception {
    mockMvc.perform(get("/students/show/{id}", 1L))
            .andExpect(status().isOk())
            .andExpect(view().name("student/showStudent"))
            .andExpect(model().attributeExists("student"))
            .andExpect(model().attribute("student",
                        hasProperty("lastName", is("Kowalski"))))
    ;
}
```

Weryfikujemy status odpowiedzi.

Ustawianie parametrów

Test dla powyższej akcji będzie wyglądał następująco:

```
@Test
public void test_show_student() throws Exception {
    mockMvc.perform(get("/students/show/{id}", 1L))
        .andExpect(status().isOk())
        .andExpect(view().name("student/showStudent"))
        .andExpect(model().attributeExists("student"))
        .andExpect(model().attribute("student",
                hasProperty("lastName", is("Kowalski"))))
    ;
}
```

Sprawdzamy czy zdefiniowany jest poprawny widok.

Ustawianie parametrów

Test dla powyższej akcji będzie wyglądał następująco:

```
@Test
public void test_show_student() throws Exception {
    mockMvc.perform(get("/students/show/{id}", 1L))
        .andExpect(status().isOk())
        .andExpect(view().name("student/showStudent"))
        .andExpect(model().attributeExists("student"))
        .andExpect(model().attribute("student",
                    hasProperty("lastName", is("Kowalski"))))
    ;
}
```

Sprawdzamy czy w modelu istnieje atrybut o nazwie **student** .

Ustawianie parametrów

Test dla powyższej akcji będzie wyglądał następująco:

```
@Test
public void test_show_student() throws Exception {
    mockMvc.perform(get("/students/show/{id}", 1L))
        .andExpect(status().isOk())
        .andExpect(view().name("student/showStudent"))
        .andExpect(model().attributeExists("student"))
        .andExpect(model().attribute("student",
                    hasProperty("lastName", is("Kowalski"))))
    ;
}
```

Sprawdzamy czy atrybut modelu posiada jedną z właściwości.

Wiele parametrów

Jeżeli akcja wymaga podania więcej niż jednego parametru, np:

```
@GetMapping("/students/params/{param1}/{param2}")
public String testParam(Model model) {
    return "student/someView";
}
```

w teście podajemy je po sobie w następujący sposób:

```
@Test
public void test_param() throws Exception {
    mockMvc.perform(get("/students/params/{param1}/{param2}", 11111L, 2222L))
    .andDo(print());
}
```

Testowanie formularza

Do naszego testowego formularza dodamy akcje umożliwiające dodanie studenta.

Wyświetlanie formularza:

```
@GetMapping("/students/add")
public String showForm(){
    return "student/addStudent";
}
```

Obsługa formularza:

```
@PostMapping("/students/add")
public String submitForm (@Valid @ModelAttribute("student") Student student,
                          BindingResult result, RedirectAttributes attributes) {
    if (result.hasErrors()) {
        return "student/addStudent";
    }
    return "redirect:/students/list";
}
```

Testowanie formularza

Dla powyższej akcji dodamy dwie metody testowe, dla poprawnego i niepoprawnego przetworzenia.

Poprawne przetworzenie:

```
@Test
public void when_save_valid_data_then_redirect() throws Exception {
    mockMvc.perform(post("/students/add").param("firstName",
                    RandomStringUtils.randomAlphabetic(10))
            .param("lastName", RandomStringUtils.randomAlphabetic(8)))
            .andExpect(redirectedUrl("/students/list"))
            .andDo(print());
}
```

Testowanie formularza

Dla powyższej akcji dodamy dwie metody testowe, dla poprawnego i niepoprawnego przetworzenia.

Poprawne przetworzenie:

```
@Test
public void when_save_valid_data_then_redirect() throws Exception {
    mockMvc.perform(post("/students/add").param("firstName",
                    RandomStringUtils.randomAlphabetic(10))
            .param("lastName", RandomStringUtils.randomAlphabetic(8)))
            .andExpect(redirectedUrl("/students/list"))
            .andDo(print());
}
```

Wywołujemy akcję, określając jej parametr żądania o nazwie **firstName**.

Testowanie formularza

Dla powyższej akcji dodamy dwie metody testowe, dla poprawnego i niepoprawnego przetworzenia.

Poprawne przetworzenie:

```
@Test
public void when_save_valid_data_then_redirect() throws Exception {
    mockMvc.perform(post("/students/add").param("firstName",
                    RandomStringUtils.randomAlphabetic(10))
            .param("lastName", RandomStringUtils.randomAlphabetic(8)))
            .andExpect(redirectedUrl("/students/list"))
            .andDo(print());
}
```

Wykorzystujemy metodę generującą losowy ciąg znaków o zadanej długości. Metoda ta jest dostępna w pakiecie **`org.apache.commons.lang3`**

Testowanie formularza

Dla powyższej akcji dodamy dwie metody testowe, dla poprawnego i niepoprawnego przetworzenia.

Poprawne przetworzenie:

```
@Test
public void when_save_valid_data_then_redirect() throws Exception {
    mockMvc.perform(post("/students/add").param("firstName",
                    RandomStringUtils.randomAlphabetic(10))
            .param("lastName", RandomStringUtils.randomAlphabetic(8)))
            .andExpect(redirectedUrl("/students/list"))
            .andDo(print());
}
```

Sprawdzamy czy zwracaną wartością jest przekierowanie do listy.

Testowanie formularza

Dla powyższej akcji dodamy dwie metody testowe, dla poprawnego i niepoprawnego przetworzenia.

Niepoprawne przetworzenie:

```
@Test
public void when_save_not_valid_data_then_show_form_again() throws Exception {
    mockMvc.perform(post("/students/add")
            .param("lastName", RandomStringUtils.randomAlphabetic(10))
            .param("otherName", RandomStringUtils.randomAlphabetic(10)))
            .andExpect(view().name("student/addStudent"));
}
```

Testowanie formularza

Dla powyższej akcji dodamy dwie metody testowe, dla poprawnego i niepoprawnego przetworzenia.

Niepoprawne przetworzenie:

```
@Test
public void when_save_not_valid_data_then_show_form_again() throws Exception {
    mockMvc.perform(post("/students/add")
            .param("lastName", RandomStringUtils.randomAlphabetic(10))
            .param("otherName", RandomStringUtils.randomAlphabetic(10)))
            .andExpect(view().name("student/addStudent"));
}
```

Wywołujemy akcję.

Testowanie formularza

Dla powyższej akcji dodamy dwie metody testowe, dla poprawnego i niepoprawnego przetworzenia.

Niepoprawne przetworzenie:

```
@Test
public void when_save_not_valid_data_then_show_form_again() throws Exception {
    mockMvc.perform(post("/students/add")
            .param("lastName", RandomStringUtils.randomAlphabetic(10))
            .param("otherName", RandomStringUtils.randomAlphabetic(10)))
            .andExpect(view().name("student/addStudent"));
}
```

Określamy parametr żądania o nazwie **`lastName`** oraz **`otherName`**. Zakładając że wymagane jest **firstName** oraz **lastName** nie posiadamy wszystkich wymaganych dannych do utworzenia obiektu **Student**.

Testowanie formularza

Dla powyższej akcji dodamy dwie metody testowe, dla poprawnego i niepoprawnego przetworzenia.

Niepoprawne przetworzenie:

```
@Test
public void when_save_not_valid_data_then_show_form_again() throws Exception {
    mockMvc.perform(post("/students/add")
            .param("lastName", RandomStringUtils.randomAlphabetic(10))
            .param("otherName", RandomStringUtils.randomAlphabetic(10)))
            .andExpect(view().name("student/addStudent"));
}
```

Sprawdzamy czy ponownie zostanie wyświetlony widok formularza dodawania.

Zadania