



SZTUKA KODU

DARIUSZ
MYDLARZ

CZEŚĆ!

Jeżeli jesteś tak jak ja, to łączą nas dwie rzeczy:

- lubisz tworzyć aplikacje w Springu,
- lubisz rozwijać swój warsztat programistyczny.

Świetnie! Bazując na moim wieloletnim doświadczeniu stworzyłem dokument, w którym prezentuję **10 Sztuczek Najlepszych Programistów w Springu.**

Dzięki nim Twój kod będzie o niebo lepszy, a Twóz zespół Cię pokocha! Mam nadzieję, że da Ci dużo wartości.

Jeśli mogę Ci jakoś pomóc, napisz do mnie
darek@sztukakodu.pl

Miłej lektury!

Dariusz Mydlarz

Autor bloga sztukakodu.pl



SZTUKA KODU

DARIUSZ
MYDLARZ

1. WSTRZYKIWANIE PRZEZ KONSTRUKTOR

Gdy tworzysz nowy komponent w Springu, zawsze, ale to zawsze korzystaj ze **wstrzykiwania przez konstruktora**.

Nie ma lepszej rzeczy, jaką możesz zrobić dla siebie i swoich kolegów w projekcie niż korzystać z finalnych pól i wstrzykiwania zależności przez konstruktora.

Dzięki temu upewniasz się, że Twój serwis domenowy zawsze zostanie stworzony w poprawny sposób i zawsze będzie miał wstrzyknięte wszystkie potrzebne komponenty.

Przy korzystaniu z pustych konstruktorów i wstrzykiwania przez settery narażasz się na możliwość stworzenia niegotowych obiektów, sprawiasz, że są one mutowalne i nie podpowiadasz programistom jak tworzyć te obiekty w testach.

Jeśli masz z tego dokumentu zapamiętać i wdrożyć jedną lekcję, niech to będzie ta.

```
@Component
public class EmailService {
    private final EmailRepository emailRepository;
    private final EmailTemplates emailTemplates;

    public EmailService(EmailRepository emailRepository, EmailTemplates emailTemplates) {
        this.emailRepository = emailRepository;
        this.emailTemplates = emailTemplates;
    }
}
```



SZTUKA KODU

DARIUSZ
MYDLARZ

2. PAKIETOWANIE PO ODPOWIEDZIALNOŚCI

W kwestii pakietowania klas w projekcie są dwie możliwości. Pakietowanie po warstwach:

- com.myapp.controllers
- com.myapp.services
- com.myapp.dao
- com.myapp.utils

Lub pakietowanie po odpowiedzialności:

- com.myapp.orders
- com.myapp.clients
- com.myapp.products
- com.myapp.auctions

Jeśli tylko zaczynasz nowy projekt, w których możesz mieć wpływ na pakietowanie, wybieraj drugą opcję. Dzięki temu dzielisz swoją architekturę według funkcji, które wykonuje, a nie według tego jakie warstwy w niej wydzielisz. To pozwala utrzymać w spójności powiązane ze sobą klasy, a także ukrywać je przed dostępem z zewnątrz przez stosowanie package-scope.

Dodatkowo dużo łatwiej na takim kodzie pracować kilku osobom, gdyż nie mieszają kodu w tych samych miejscach.

Jeśli jeszcze nie układłeś kodu w taki sposób, spróbuj i zobacz czy całość nie funkcjonuje lepiej.



SZTUKA KODU

DARIUSZ
MYDLARZ

```
article/
└── ArticleActionsEndpoint.java
└── ArticleContentValidationEndpoint.java
└── ArticleCrudEndpoint.java
└── UpdateArticleDtoVersusPathParameterValidator.java
└── domain
    ├── Article.java
    ├── ArticleConfiguration.java
    └── ArticleFacade.java
    ├── ArticleFactory.java
    ├── ArticlePreviewer.java
    ├── ArticlePublisher.java
    ├── ArticleRejector.java
    ├── ArticleRemover.java
    ├── ArticleRepository.java
    ├── ArticleRules.java
    ├── ArticleStatus.java
    ├── ArticleSubmitter.java
    ├── ArticleUnpublisher.java
    ├── ArticleUpdater.java
    ├── Category.java
    └── ArticleConfiguration.java
        @Configuration
        class ArticleConfiguration {
            @Bean
            ArticleFacade articleFacade() {
                ArticleUpdater updater =
                ArticleSubmitter submitter =
                ArticleRemover remover =
                ArticleRejector rejector =
                ArticleUnpublisher unpublisher =
                ArticlePublisher publisher =
                ArticlePreviewer previewer =
                return new ArticleFacade(
                    ...
                );
            }
        }
    
```

3. WYSTAWIANIE FASAD

Jeśli już podzieliłeś klasy swojej aplikacji według odpowiedzialności możesz pójść krok dalej i zastosować fasady.

Zamiast wystawiać publicznie wszystkie swoje klasy, wystawiasz tylko fasadę (np. **ArticleFacade**) i to tylko z nia komunikują się inne moduły.

Dzięki temu zamykasz swój moduł przed światem - abstrakcje nie wyciekają na zewnątrz - i tylko udostępniasz jeden interfejs za pomocą, którego inne elementy aplikacji mogą się z nim komunikować.

W listingu po lewej widać, że tylko ArticleFacade jest klasą publiczną, a na dole jak ją utworzyć. Zauważ, że tylko fasada jest beanem tworzonym w kontekście Springa, a całą resztę można łatwo stworzyć ręcznie.



SZTUKA KODU

DARIUSZ
MYDLARZ

4. UŻYWANIE PROFILI

Profile w Springu dostarczają mechanizm segregacji elementów konfiguracji aplikacji.

Za ich pomocą możesz w łatwy sposób zdecydować jak Twoja aplikacja ma działać na środowisku deweloperskim, jak na stagingu lub jak na produkcji.

Co więcej możesz także zmieniać konfigurację do testów i wtedy łączyć się np. do bazy danych w pamięci H2, podczas gdy przy zwykłym uruchomieniu chcesz się łączyć do normalnej bazy danych.

Uruchamiając aplikację decydujesz jakie profile mają być aktywne i zgodnie z tym Spring będzie używał innych elementów konfiguracji.

```
@Component
@Profile("dev")
public class DevDatasourceConfig implements DatasourceConfig {
    @Override
    public void setup() {
        System.out.println("Setting up datasource for DEV environment. ");
    }
}

@Component
@Profile("production")
public class ProductionDatasourceConfig implements DatasourceConfig {
    @Override
    public void setup() {
        System.out.println("Setting up datasource for PRODUCTION environment. ");
    }
}

public class SpringProfilesTest {
    @Autowired
    DatasourceConfig datasourceConfig;

    public void setupDatasource() {
        datasourceConfig.setup();
    }
}
```



SZTUKA KODU

DARIUSZ
MYDLARZ

5. ELIMINACJA MAGICZNYCH LICZB

Druga najważniejsza lekcja po wstrzykiwaniu przez konstruktor. Nigdy nie stosuj magicznych liczb w swoich komponentach.

Zawsze staraj się przekazywać liczby konfiguracyjne z zewnątrz.

Dzięki temu będziesz mógł łatwiej sterować zachowaniem swoich klas oraz przenieść te wartości do plików z konfiguracją.

Korzystając z tego mechanizmu w połączeniu z profilami z punktu wyżej możesz w łatwy sposób modyfikować zachowanie swojej aplikacji na różnych środowiskach.

Przykładowo możesz ustawić inne timeouty na requesty HTTP na produkcji a inne na środowisku testowym.

Nigdy nie stosuj magicznych liczb. Zawsze przekazuj komponentom konfigurację z zewnątrz.

```
class HttpClient {  
    private final CloseableHttpClient client;  
  
    HttpClient(HttpClientConfig config) {  
        RequestConfig config = RequestConfig.custom()  
            .setConnectTimeout(config.timeoutMillis())  
            .setConcurrency(config.concurrencyLevel())  
            .setKeepAlive(config.keepAlive())  
            .build();  
        client =  
            HttpClientBuilder.create().setDefaultRequestConfig(config).build();  
    }  
}
```



SZTUKA KODU

DARIUSZ
MYDLARZ

6. UNIKALNE ID W ENCJACH

Jeśli korzystasz ze Springa, to nie sposób nie korzystać z JPA do komunikacji z bazą danych.

Jedną z najtrudniejszych rzeczy jest definiowanie unikalności encji. Najprostszym sposobem byłoby stosowanie pola **id** pochodzącego z bazy danych.

Ale to pole nie jest stworzone dopóki encja nie jest zapisana do bazy danych.

Dlatego rozwiązaniem jest by jeszcze przed persystencją danej encji generować jej unikalne **uuid**, które będzie do niej przypisane już na zawsze. Dzięki temu zarówno przed zapisaniem do bazy jak i po będziemy mieli dostępne pole, które jednoznacznie określi nam czy mamy do czynienia z tym samym, czy z innym obiektem.

W tym celu możemy stworzyć sobie abstrakcyjną klasę i używać jej w każdej encji poprzez dziedziczenie.

```
@MappedSuperclass
public abstract class BaseEntity implements Serializable {

    @Id @GeneratedValue
    private Long id;

    private String uuid = UUID.randomUUID().toString();

    public int hashCode() {
        return Objects.hash(uuid);
    }

    public boolean equals(Object that) {
        return this == that || that instanceof BaseEntity
               && Objects.equals(uuid, ((BaseEntity) that).uuid);
    }
}
```



SZTUKA KODU

DARIUSZ
MYDLARZ

7. TESTOWANIE Z MOCKMVC

Pisanie testów to jedna z najważniejszych czynności pracy programisty.

Dzięki nim wiemy, czy wprowadzając zmiany do systemu nie psujemy innych części kodu.

Oprócz testów unitowych czy integracyjnych powinniśmy też pisać testy webowe, które przetestują jak zachowują się endpointy w naszej aplikacji.

Jedną z najlepszych bibliotek do tego celu jest **MockMvc**, za pomocą której w jasny i klarowny sposób - niczym pisząc zdania - możemy stworzyć testy do endpointów wystawianych przez aplikację.

```
@RunWith(SpringRunner.class)
@WebMvcTest(GreetingController.class)
public class WebMockTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private GreetingService service;

    @Test
    public void greetingShouldReturnMessageFromService() throws Exception {
        when(service.greet()).thenReturn("Hello Mock");
        this.mockMvc
            .perform(get("/greeting"))
            .andDo(print())
            .andExpect(status().isOk())
            .andExpect(content().string(containsString("Hello Mock")));
    }
}
```



SZTUKA KODU

DARIUSZ
MYDLARZ

sztukakodu.pl

8. TESTOWANIE Z WIREMOCK

Testowanie własnego API mamy już za sobą z pomocą MockMvc.

Potrzebujemy jednak też przetestować komunikację z zewnętrznymi serwisami.

O ile kilka testów integracyjnych ma sens, o tyle wiele lepiej jest zestubować zewnętrzne serwisy.

Dzięki temu nasze testy będą wykonywać się o wiele szybciej, oraz będziemy niezależni od statusu zewnętrznej aplikacji.

Narzędziem, które ja i wielu znanych mi programistów korzysta jest **Wiremock**.

Pozwala on za pomocą deklaratywnego języka zdefiniować jak powinien odpowiadać zewnętrzny serwer i tym samym pomoże nam w odpowiednim przetestowaniu komunikacji z nim.

```
@Test
public void exampleTest() {
    stubFor(get(urlEqualTo("/my/resource"))
        .withHeader("Accept", equalTo("text/xml"))
        .willReturn(aResponse()
            .withStatus(200)
            .withHeader("Content-Type", "text/xml")
            .withBody("<response>Some content</response>")));

    Result result = myHttpServiceCallingObject.doSomething();

    assertTrue(result.wasSuccessful());

    verify(postRequestedFor(urlMatching("/my/resource/[a-z0-9]+"))
        .withRequestBody(matching(".*<message>1234</message>.*"))
        .withHeader("Content-Type", notMatching("application/json")));
}
```

Źródło: <http://wiremock.org/docs/getting-started/>



SZTUKA KODU

DARIUSZ
MYDLARZ

9. UŻYWANIE LOMBOKA

O Javie mówi się wiele złego w związku z dużą ilością tak zwanego **boilerplate-u**, czyli kodu, który niewiele wnosi do funkcjonalności aplikacji.

Jednym z takich problemów jest konieczność generowania getterów i setterów na potrzeby zewnętrznych bibliotek - np. Hibernate-a.

Na szczęście na każdy problem jest rozwiązanie. Jedną z najczęściej stosowanych w ostatnich latach bibliotek jest **Lombok**, który za pomocą adnotacji - takich jak na przykład **@Data** - jest w stanie generować odpowiednie metody bez napisania przez nas nawet linijki kodu.

```
@Data
public class DataExample {
    private final String name;
    @Setter(AccessLevel.PACKAGE)
    private int age;
    private double score;
    private String[] tags;

    @ToString(includeFieldNames=true)
    @Data(staticConstructor="of")
    public static class Exercise<T> {
        private final String name;
        private final T value;
    }
}
```



SZTUKA KODU

DARIUSZ
MYDLARZ

10. KORZYSTANIE Z FAKEÓW

Testowanie ważniejszych części kodu często wymaga stworzenia całego drzewa obiektów, które ze sobą współdziałyają.

Do niektórych testów wystarczą nam jednak wydmuszki prawdziwych obiektów.

Jedną z ciekawszych technik dostarczania takich zależności jest stosowanie **Fake-ów**.

Zamiast ręcznie rzeźbić potencjalne odpowiedzi w frameworkach mockujących jak Mockito, dużo lepiej jest po prostu dostarczyć sztuczną implementację pożądanego interfejsu.

Dzięki temu przykładowo dostęp elementów z bazy danych można w testach przygotować w taki prosty i przyjemny sposób jak w listingu poniżej.

```
public class FakeAccountRepository implements AccountRepository {  
  
    Map<User, Account> accounts = new HashMap<>();  
  
    public FakeAccountRepository() {  
        this.accounts.put(new User("john@bmail.com"), new UserAccount());  
        this.accounts.put(new User("boby@bmail.com"), new AdminAccount());  
    }  
  
    String getPasswordHash(User user) {  
        return accounts.get(user).getPasswordHash();  
    }  
}
```