

Spring framework

2

Cele szkolenia

Spring framework

- Praktyczne poznanie Spring framework (rozbudowana część warsztatowa)
- Zrozumienie architektury i sposobu działania frameworku
- Nabycie dobrych praktyk związanych m.in. z testowaniem, debuggowaniem czy utrzymaniem dużego projektu
- Poznanie metod integracji z innymi rozwiązaniami (Java EE oraz Angular)

3

Spring framework

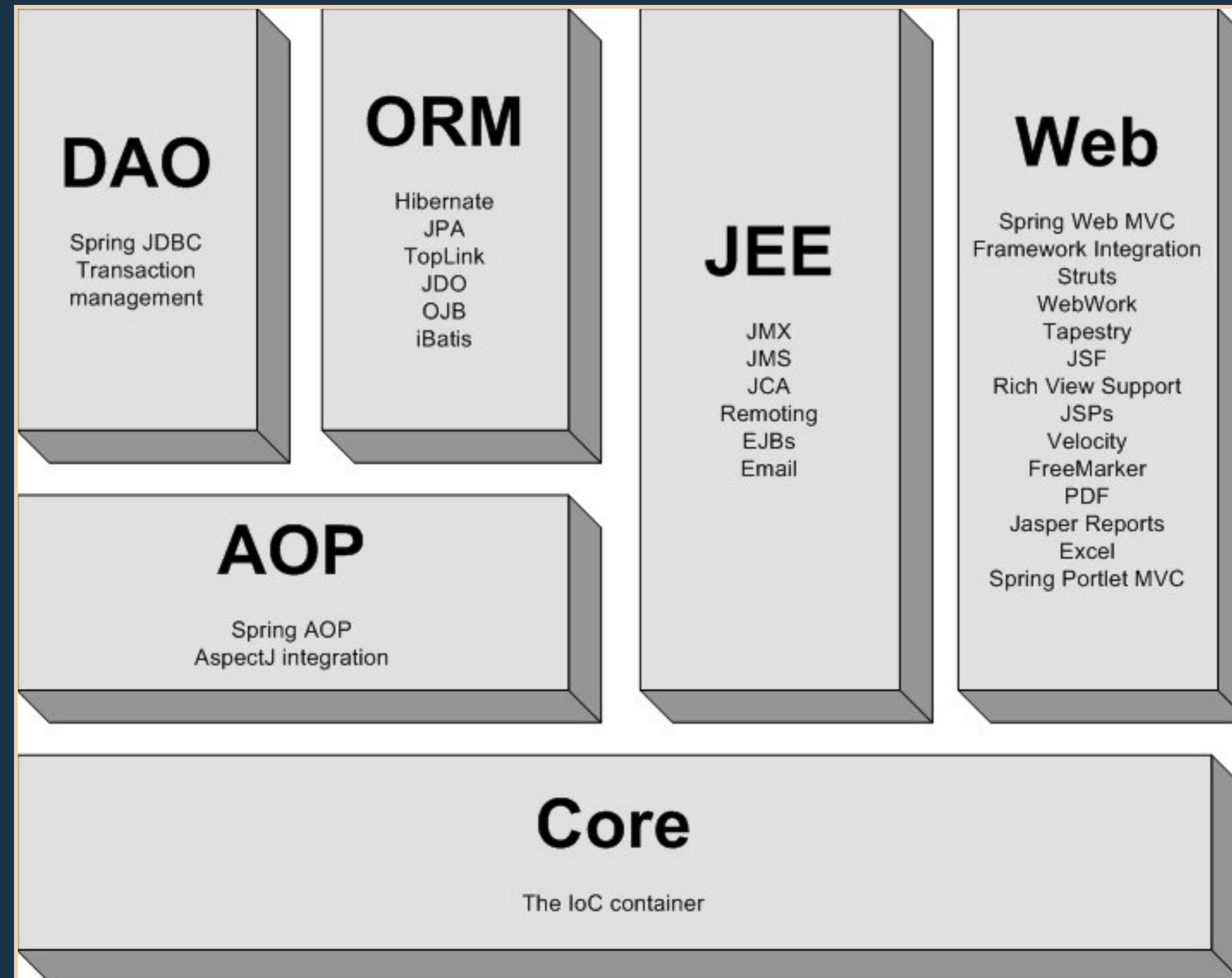
Spring framework

- Jeden z najpopularniejszych i najbardziej uniwersalnych frameworków języka Java
- Wykorzystuje mechanizm wstrzykiwania zależności oraz paradygmat programowania aspektowego
- Pozwala na zastosowanie podejścia komponentowego w oparciu o lekkie obiekty typu POJO (Plain Old Java Object)
- Promuje programowanie przez interfejsy oraz użycie najlepszych praktyk programistycznych
- Umożliwia integrację z najpopularniejszymi technologiami i frameworkami

4

Architektura Spring framework

Spring framework



5

Wstrzykiwanie zależności (ang. dependency injection)

Spring framework

- Programy tworzone w języku Java realizują zadania poprzez zbiór współpracujących ze sobą obiektów, co wynika bezpośrednio z paradygmatu programowania obiektowego
- Obiekty posiadają wzajemne zależności, które komplikują utrzymanie, modyfikowanie i testowanie kodu
- Spring pozwala na rozluźnienie powiązań między obiektami poprzez automatyczne zarządzanie zależnościami

6

Programowanie aspektowe (ang. aspect oriented programming)

Spring framework

- Uzupełnia paradygmat programowania obiektowego
- Umożliwia oddzielenie logiki biznesowej od dodatkowych zadań pobocznych takich jak np.: transakcje, logowanie, bezpieczeństwo

7

Strategia Spring framework

Spring framework

- Lekkie i nieinwazyjne programowanie z użyciem obiektów typu POJO
- Niskie sprzężenie uzyskiwane za pomocą wstrzykiwania zależności i programowania przez interfejsy
- Redukcja kodu typu „boilerplate” przez zastosowanie aspektów i szablonów
- Programowanie deklaratywne dzięki aop i przyjętym konwencjom

8

Kontener inwersji kontroli (ang. inversion of control)

Spring framework

- Programy tworzone w języku Java składają się z obiektów współpracujących w celu realizacji założonych zadań
- Tradycyjne podejście do definiowania zależności między obiektami prowadzi do kodu trudnego w utrzymaniu i testowaniu (obiekty często wykonują więcej niż powinny i są silnie sprzężone)
- Aplikacje oparte o Spring Framework wykorzystują kontener IoC jako implementację mechanizmu wstrzykiwania zależności
- Kontener odpowiada za tworzenie, konfigurowanie i zarządzanie cyklem życia beanów, a także dostarcza im wszystkie niezbędne usługi

9

Typy kontenerów i ich implementacje

Spring framework

- Spring dostarcza kilka implementacji kontenerów, które można sklasyfikować w następujący sposób:
 - Wywodzące się z interfejsu `org.springframework.beans.factory.BeanFactory`, zapewniają wsparcie dla mechanizmu wstrzykiwania zależności
 - Wywodzące się z interfejsu `org.springframework.context.ApplicationContext`, rozszerzają funkcjonalność kontenerów typu `BeanFactory`, dostarczają dodatkowe usługi takie jak: propagacja i obsługa zdarzeń, wsparcie dla internacjonalizacji, automatyczne rejestrowanie beanów specjalnych, wczesna inicjalizacja beanów o zasięgu singleton
- Najbardziej popularne implementacje kontenerów typu `ApplicationContext` to:
 - `ClassPathXmlApplicationContext`
 - `FileSystemXmlApplicationContext`
 - `XmlWebApplicationContext`
 - `AnnotationConfigApplicationContext`

- Stworzenie opisu beanów wchodzących w skład aplikacji (rejestr XML, adnotacje, Java-configuration)
- Wybór implementacji i stworzenie instancji kontenera
- Pozyskanie referencji do beanów zarządzanych przez kontener

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean id="inMemoryRepository" class="pl.szkolenie.repository.InMemoryRepository">
    <!-- konfiguracja -->
  </bean>

  <bean id="..." class="...">
    <!-- konfiguracja -->
  </bean>

</beans>
```

12

Tworzenie instancji kontenera i pobieranie beanów

Spring framework

```
package pl.szkolenie;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml")) {
            BankRepository bankRepository = ctx.getBean("inMemoryRepository", BankRepository.class);
        }
    }
}
```

- Wszystkie beany zarządzane w ramach kontenera posiadają przynajmniej jeden, unikalny identyfikator
- Identyfikatory nadawane są za pomocą atrybutu `id` i/lub `name` elementu `bean` lub automatycznie przez kontener np. w przypadku beanów wewnętrznych
- Wartość atrybutu `id` podlega walidacji zgodnie ze specyfikacją standardu XML dlatego jego użycie jest preferowane
- Atrybut `name` pozwala określić wiele identyfikatorów (oddzielonych przecinkiem, spacją lub średnikiem), które mogą zawierać znaki specjalne np. „/”
- Stosowane nazwy powinny mieć charakter opisowy i zwyczajowo pisane są z małej litery zgodnie z konwencją camel-case

- Definicja beana jest receptą pozwalającą na stworzenie jednego lub więcej obiektów
- W większości przypadków instancje tworzone są z pomocą mechanizmu refleksji, poprzez konstruktor klasy określonej atrybutem **class**
- W przypadku statycznych klas wewnętrznych jako wartość argumentu **class** wprowadzić należy pełną, binarną nazwę klasy np.:

```
<bean id="keyGenerator" class="pl.szkolenie.repository.KeyGenerator"/>
```


15

Metody fabrykujące

Spring framework

- Framework Spring pozwala używać statycznej metody fabrykującej do tworzenia instancji beanów
- Typ zwracanego obiektu nie musi być tożsamy z typem klasy zawierającej metodę fabrykującą, a tym samym z typem określonym przez atrybut **class**

```
<bean id="identyfikator" class="nazwaKlasyZawierającaFabrykę" factory-method="metodaFabrykująca"/>
```

- Analogicznie istnieje możliwość użycia fabryk instancyjnych - metod fabrykujących zdefiniowanych w ramach innych beanów

```
<bean id="identyfikator" factory-bean="beanZawierającyFabrykę" factory-method="metodaFabrykująca"/>
```

- Wstrzykiwanie ewentualnych argumentów w obu powyższych przypadkach odbywa się analogicznie jak wstrzykiwanie przez konstruktor omówione w dalszej części modułu

- Odbywa się przy użyciu znaczników `<constructor-arg/>` i/lub `<property/>` zagnieżdżanych wewnątrz znacznika `<bean/>`
- W przypadku wartości podawanych w postaci tekstu (typy prymitywne, String oraz inne) stosuje się atrybut `value`, a dla zależności będących referencją do innego beana atrybut `ref`
- Konwersja wartości podawanych w formie tekstu odbywa się za pomocą specjalnych obiektów typu `Converter` (wbudowanych lub zdefiniowanych przez programistę)

17

Wstrzykiwanie zależności przez konstruktor

Spring framework

- Domyślnie argumenty przekazywane są w kolejności w jakiej zostały zdefiniowane

```
package x.y;
    public class Foo {
        public Foo(Bar bar, Baz baz) {
            // ...
        }
    }
}
```

```
<beans>
    <bean id="foo" class="x.y.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
    </bean>
    <bean id="bar" class="x.y.Bar"/>
    <bean id="baz" class="x.y.Baz"/>
</beans>
```

18

Wstrzykiwanie zależności przez konstruktor

Spring framework

- W przypadku argumentów będących typami prostymi należy określić dodatkowo atrybut **type**

```
package examples;
```

```
public class ExampleBean {  
    private int years;  
    private String ultimateAnswer;  
    public ExampleBean(int years, String ultimateAnswer) {  
        this.years = years;  
        this.ultimateAnswer = ultimateAnswer;  
    }  
}
```

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <constructor-arg type="int" value="7500000"/>  
    <constructor-arg type="java.lang.String" value="42"/>  
</bean>
```

19

Wstrzykiwanie zależności przez konstruktor

Spring framework

- Alternatywą pozwalającą na rozróżnienie argumentów jest użycie argumentu **index**, którego wartości rozpoczynają się od 0

```
package examples;
```

```
public class ExampleBean {  
    private int years;  
    private String ultimateAnswer;  
    public ExampleBean(int years, String ultimateAnswer) {  
        this.years = years;  
        this.ultimateAnswer = ultimateAnswer;  
    }  
}
```

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <constructor-arg index="0" value="7500000"/>  
    <constructor-arg index="1" value="42"/>  
</bean>
```

- Rozpoczynając od wersji Spring 3.0 istnieje możliwość wykorzystania atrybutu **name**, którego wartość przybiera poszczególne nazwy argumentów

```
package examples;
```

```
public class ExampleBean {  
    private int years;  
    private String ultimateAnswer;  
    public ExampleBean(int years, String ultimateAnswer) {  
        this.years = years;  
        this.ultimateAnswer = ultimateAnswer;  
    }  
}
```

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <constructor-arg name="years" value="7500000"/>  
    <constructor-arg name="ultimateAnswer" value="42"/>  
</bean>
```

- Odbywa się po stworzeniu beana i uprzednim wywołaniu konstruktora
- Może być łączone razem z wstrzykiwaniem przez konstruktor

```
package examples;
```

```
public class ExampleBean {  
    private int length;  
    public setLength(int length) {  
        this.length = length;  
    }  
}
```

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <property name="length" value="42"/>  
</bean>
```

- Beany wewnętrzne to beany definiowane w ramach znaczników `<constructor-arg/>` lub `<property/>`
- Pozwalają na zdefiniowanie zależności w miejscu jej deklaracji
- Nie można im przypisać nazwy (kontener ją ignoruje i generuje własną), a ich zasięg jest zawsze typu `prototype`

```
<bean id="outer" class="examples.PhoneBook">
  <property name="target">
    <bean class="examples.Person">
      <property name="name" value="Jan Kowalski"/>
      <property name="age" value="12"/>
    </bean>
  </property>
</bean>
```


- Podczas tworzenia programu może dojść do sytuacji w której bean będzie zależał od istnienia innego beana, jednak nie będzie posiadał do niego bezpośredniej referencji
- Zależności tego typu definiuje się przy pomocy atrybutu `depends-on` (podawane wartości mogą być oddzielone przecinkami, spacjami lub średnikami)
- Określona zależność ma wpływ na kolejność tworzenia oraz niszczenia beanów

```
<bean id="beanOne" class="beanOne" depends-on="beanTwo"/>
```

```
<bean id="beanTwo" class="beanTwo" />
```

- Kontener dostarcza możliwość automatycznego wstrzykiwania zależności, bez konieczności używania atrybutu ref

```
<bean id="nazwa" class="klasa" autowire="tryb"/>
```

- Istnieje możliwość ustawienia wstrzykiwania automatycznego na poziomie globalnym

```
<beans default-autowire="tryb">
```

- Autowiązanie działa w następujących trybach
 - **no** - bez automatycznego wstrzykiwania, wartość domyślna
 - **byName** - aby nastąpiło wstrzyknięcie id beana musi być identyczne jak nazwa właściwości
 - **byType** - kandydat do wstrzyknięcia poszukiwany jest na podstawie typu; w kontenerze może istnieć tylko jeden bean poszukiwanego typu; wstrzyknięcie następuje poprzez metody set
 - **constructor** - analogiczne do byType tylko wstrzyknięcie następuje poprzez konstruktor

- Bez automatycznego wstrzykiwania zależności

```
<bean id="car" class="pl.szukolenie.impl.CarImpl">  
  <property name="engine" ref="engine"/>  
</bean>
```

```
<bean id="engine" class="pl.szukolenie.impl.EngineImpl"/>
```

- Z automatycznym wstrzykiwaniem zależności

```
<bean id="car" class="pl.szukolenie.impl.CarImpl" autowire="byName"/>  
<bean id="engine" class="pl.szukolenie.impl.EngineImpl"/>
```

- Definicja beana może zawierać sporo informacji takich jak: zależności wstrzykiwane przez konstruktor i settery, konfigurację metody inicjalizującej i sprzątającej, tryb autowiązania i wiele innych
- Spring pozwala na dziedziczenie konfiguracji beanów na poziomie XML
- Rozwiązanie pozwala znacznie zmniejszyć ilość niezbędnej konfiguracji jednak może prowadzić do zmniejszenia jej czytelności
- Użycie atrybutu **abstract** uniemożliwia tworzenie instancji beana. Atrybut jest obowiązkowy w przypadku gdy nie zdefiniowano atrybutu **class**

```
<bean id="inheritedTestBean" abstract="true" class="org.springframework.beans.TestBean">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>
<bean id="inheritsWithDifferentClass" parent="inheritedTestBean" init-method="initialize">
    <property name="name" value="override"/>
</bean>
```

- Spring definiuje pięć standardowych zasięgów w których mogą występować beany: **singleton**, **prototype**, **request**, **session**, **global session** (trzy ostatnie wymagają dodatkowej konfiguracji)
- Istnieje możliwość definiowania zasięgów niestandardowych
- Domyślny zasięg to **singleton**
- Zmiana zasięgu odbywa się poprzez atrybut **scope**

```
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>
```


- Użycie atrybutu `init-method` dla elementu `bean` pozwala na zdefiniowanie metody inicjalizującej wywoływanej po stworzeniu instancji i wstrzyknięciu jej zależności

```
<bean id="myService" class="pl.szkozenie.Service" init-method="init"/>
```

```
public class Service {  
    public void init() {  
        //...  
    }  
}
```

- Zaleca się używanie tej samej nazwy metody inicjalizującej dla każdego beana; w takim przypadku można ją zdefiniować globalnie na poziomie elementu `beans`:

```
<beans default-init-method="init">  
    ...  
</beans>
```

- Użycie atrybutu `destroy-method` dla elementu `bean` pozwala na zdefiniowanie metody finalizującej wywoływanej przed zniszczeniem instancji (nie działa dla prototypów)

```
<bean id="myService" class="pl.szkozenie.Service" destroy-method="destroy"/>
```

```
public class Service {  
    public void destroy() {  
        //...  
    }  
}
```

- Zaleca się używanie tej samej nazwy metody finalizującej dla każdego beana; w takim przypadku można ją zdefiniować globalnie na poziomie elementu `beans`:

```
<beans default-destroy-method="destroy">  
    ...  
</beans>
```


- Interfejs **BeanPostProcessor** pozwala na zdefiniowanie metod typu callback wykonywanych przez kontener przed i po inicjalizacji każdego z beanów
- W celu zarejestrowania post procesora należy zdefiniować go jako jeden z beanów w kontekście kontenera
- Określenie kolejności wykonywania obiektów typu **BeanPostProcessor** jest możliwe poprzez implementację interfejsu **Ordered** i ustalenie odpowiedniej wartości pola **order**

```
package pl.szkolenie;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;

public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }

    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("Bean '" + beanName + "' created : " + bean.toString());
        return bean;
    }
}
```

- Interfejs **BeanFactoryPostProcessor** pozwala na stworzenie obiektów mających dostęp do rejestru beanów
- Mogą one zarówno czytać jak i dokonywać modyfikacji konfiguracji beanów zanim jakikolwiek z nich zostanie stworzony
- W celu zarejestrowania post procesora należy zdefiniować go jako jeden z beanów w kontekście kontenera
- Określenie kolejności wykonywania obiektów typu **BeanFactoryPostProcessor** jest możliwe poprzez implementację interfejsu **Ordered** i ustalenie odpowiedniej wartości pola **order**

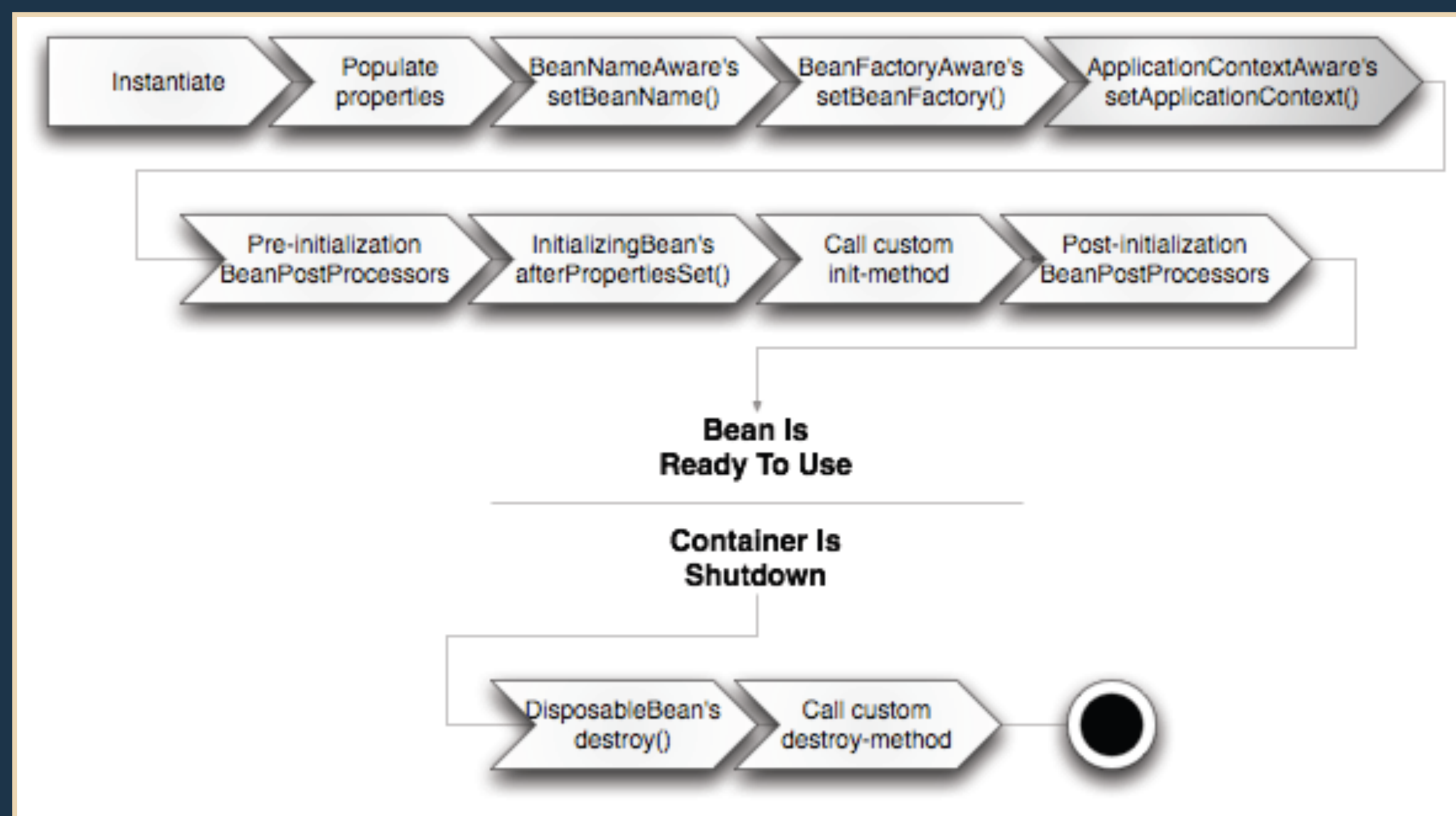
33

Obiekty specjalne typu BeanFactoryPostProcessor

Spring framework

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
    <property name="locations" value="classpath:com/foo/jdbc.properties"/>  
</bean>
```

```
<bean id="dataSource" destroy-method="close" class="org.apache.commons.dbcp.BasicDataSource">  
    <property name="driverClassName" value="${jdbc.driverClassName}"/>  
    <property name="url" value="${jdbc.url}"/>  
    <property name="username" value="${jdbc.username}"/>  
    <property name="password" value="${jdbc.password}"/>  
</bean>
```



- Paradygmat programowania pozwalający odseparować logikę biznesową od funkcjonalności pobocznych (ang. cross-cutting concerns)
- Logowanie, bezpieczeństwo, obsługa transakcji oraz inne funkcjonalności zamykane są w ramach specjalnych klas nazywanych aspektami, a następnie deklaratywnie aplikowane do wybranych modułów aplikacji (bez konieczności ich modyfikacji)
- Programowanie aspektowe przyczynia się do:
 - Zwiększenia czytelności kodu - usługi skupiają się wyłącznie na realizacji logiki biznesowej
 - Zmniejszenia kosztów utrzymania i modyfikacji kodu - funkcjonalności poboczne zamknięte są w ramach aspektów, a nie rozsiane po całej aplikacji

- **Aspect** - moduł/jednostka skupiająca się na realizacji funkcjonalności używanej w wielu miejscach aplikacji
- **Join point** - punkt w wykonywanym programie, w którym potencjalnie można dołączyć aspekt
- **Advice** - akcja wykonywana przez aspekt w określonym punkcie joint point
- **Pointcut** - wyrażenie określające miejsca (wybrane join points), w których powinna być dodana funkcjonalność zaimplementowana w aspekcie
- **Target object** - obiekt docelowy do którego dodawana jest funkcjonalność aspektu
- **Weaving** - proces dodawania aspektów do obiektów docelowych

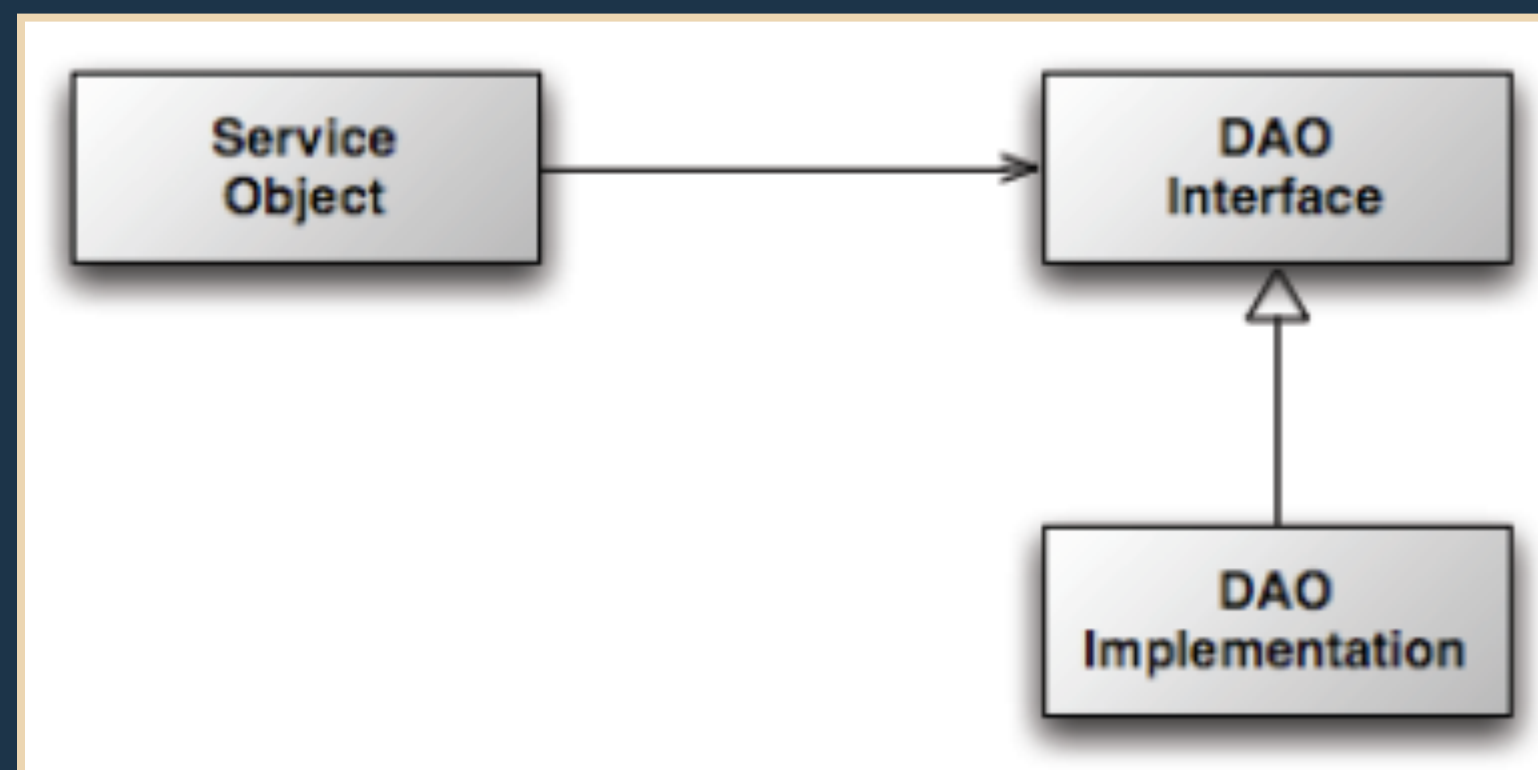
- **Before advice** - implementowana funkcjonalność ma pierwszeństwo przed wykonaniem metody docelowej
- **After advice** - implementowana funkcjonalność jest realizowana po wykonaniu metody docelowej niezależnie od zwróconego rezultatu
- **After returning advice** - implementowana funkcjonalność jest realizowana tylko po poprawnym zakończeniu metody docelowej
- **After throwing advice** - implementowana funkcjonalność jest realizowana tylko po wyrzuceniu wyjątku z metody docelowej
- **Around advice** - implementowana funkcjonalność opakowuje całe wykonanie metody docelowej

- Proces dodawania aspektów może być realizowany w różnych fazach cyklu życia obiektów docelowych:
 - Przy kompilacji
 - Podczas ładowania klasy do maszyny wirtualnej
 - W czasie wykonywania programu (obiekty proxy)

- Zaimplementowany w języku Java
- Używa elementów składni AspectJ
- Umożliwia definiowanie aspektów przy użyciu plików XML lub adnotacji
- Pozwala dołączać aspekty wyłącznie do metod publicznych
- Dla każdego z obiektów docelowych tworzy automatycznie obiekt proxy

Obiekty typu DAO (ang. data access objects)

Spring framework



- Spring dostarcza mechanizm konwersji wyjątków specyficznych dla najpopularniejszych technologii utrwalania do spójnej hierarchii wyjątków opartej o typ `DataAccessException`
- Wyrzucane wyjątki nie muszą być przechwytywane w blokach catch ponieważ wywodzą się z typu `RuntimeException`

- Spring separuje stałe i zmienne elementy procesu dostępu do danych
- Klasy typu Templates odpowiadają za realizację typowych, powtarzających się operacji (otwieranie i zamykanie połączenia, obsługa wyjątków, zarządzanie transakcjami), a także dostarczają api upraszczające operowanie na danych
- Klasy typu Callback zawierają zmienną logikę dostępu do danych (tworzenie wyrażeń statement, podstawianie parametrów, mapowanie wyników)
- Framework Spring dostarcza kilka implementacji szablonów zależnych od technologii utrwalania, a także klas typu DAO support mogących służyć jako baza dla obiektów dostępowych

- Wszystkie szablony i klasy typu DAO support oczekują skonfigurowanego połączenia z bazą danych w postaci obiektu DataSource
- Spring oferuje kilka opcji konfiguracji źródła danych m.in.:
 - Z użyciem lokalnego sterownika JDBC
 - Poprzez pobranie referencji przy użyciu usługi JNDI
 - Z wykorzystaniem puli połączeń

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
    <property name="url" value="jdbc:hsqldb:hsqldb://localhost/myDatabase" />
    <property name="username" value="user" />
    <property name="password" value="123" />
</bean>
```

```
<context:property-placeholder location="classpath:jdbc.properties"/>
```

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="url" value="${database.url}" />
    <property name="driverClassName" value="${database.driver}" />
    <property name="username" value="${database.user}" />
    <property name="password" value="${database.password}" />
</bean>
```

- Znacznik `<jee:jndi-lookup>` pozwala pobrać dowolny zasób udostępniony w ramach usługi JNDI i zwrócić w formie beana
- Atrybut `jndi-name` wskazuje nazwę zasobu w rejestrze

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/mysql" />
```

- Projekt Jakarta Commons Database Connection Pools (DBCP)

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
  <property name="url" value="jdbc:hsqldb:hsqldb://localhost/myDatabase" />
  <property name="username" value="user" />
  <property name="password" value="123" />
  <property name="initialSize" value="4" />
  <property name="maxActive" value="8" />
</bean>
```

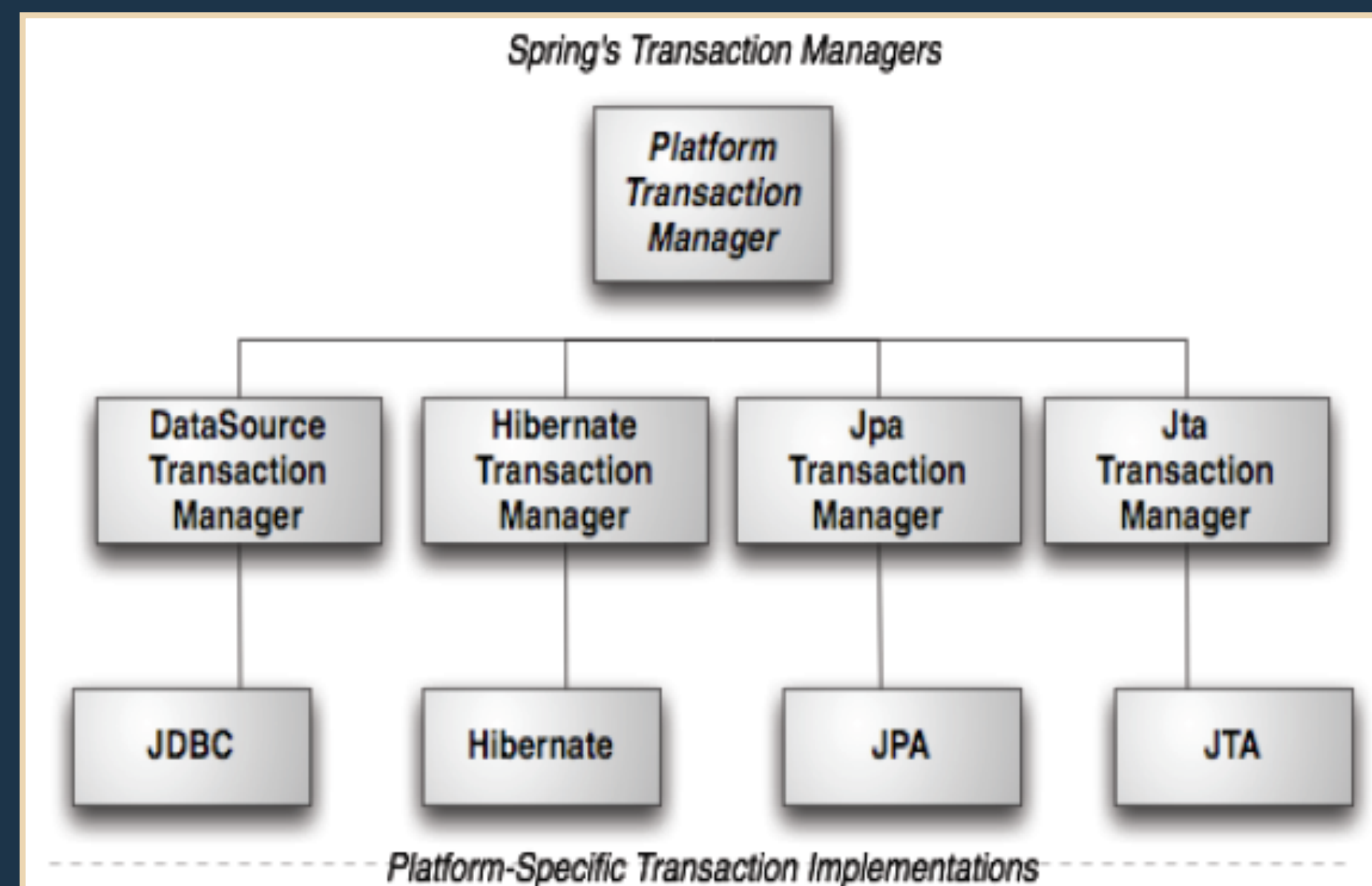
```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
    <constructor-arg ref="dataSource" />
</bean>
```

```
<bean id="myDao" class="pl.szkolenie.myDao">
    <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

```
public class myDAO implements DAO {
    private SimpleJdbcTemplate jdbcTemplate;
    public void setJdbcTemplate(SimpleJdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

- Transakcja to zbiór czynności podejmowanych w celu realizacji założonego zadania, z których każda musi zostać zakończona w sposób prawidłowy (w innym przypadku wprowadzone zmiany są wycofywane)
- Transakcje są mechanizmem pozwalającym na zachowanie integralności danych na których operuje jednocześnie wiele użytkowników/systemów
- Prawidłowa transakcja powinna spełniać kryteria ACID czyli być:
 - atomowa - każda z czynności składających się na realizowane zadanie musi zostać wykonana bezbłędnie i w całości w przeciwnym przypadku transakcja jest przerywana, a wykonane zmiany są cofane
 - spójna - składowe systemu muszą zachować integralność w czasie i po zakończeniu transakcji
 - izolowana - dane używane podczas transakcji nie mogą być wykorzystywane przez inne elementy systemu w czasie trwania transakcji
 - trwała - zmiany wprowadzone podczas transakcji muszą zostać utrwalone w pamięci fizycznej

- Framework Spring nie zarządza transakcjami w sposób bezpośredni lecz dostarcza zbiór gotowych menadżerów transakcji (ang. transaction managers), które delegują odpowiedzialność do konkretnych implementacji mechanizmów transakcyjnych, zależnych od używanej platformy (JDBC, Hibernate, JPA, JTA)



```
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
    <property name="dataSource" ref="dataSource"/>  
</bean>
```

```
<bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">  
    <property name="sessionFactory" ref="sessionFactory"/>  
</bean>
```

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">  
    <property name="entityManagerFactory" ref="entityManagerFactory" />  
</bean>
```

```
<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager">  
    <property name="transactionManagerName" value="java:/TransactionManager" />  
</bean>
```

- **Poziom izolacji (ang. isolation level)** - definiuje stopień, w jakim transakcja ma dostęp do niezatwierdzonych wyników działania innej transakcji pracującej na tych samych danych
- **Propagacja (ang. propagation)** - określa sposób propagowania kontekstu transakcji w ramach wywoływanych metod
- **Czas ważności (ang. timeout)** - wskazuje dopuszczalny czas wykonywania transakcji
- **Status tylko do odczytu (ang. readonly status)** - informuje czy mogą być stosowane mechanizmy optymalizacyjne związane z faktem, że dane są tylko odczytywane
- **Reguły wycofywania (ang. rollback rules)** - opisuje jakie działania powinny być podjęte przez mechanizm transakcyjny w przypadku wystąpienia określonych wyjątków

- **DEFAULT** - zgodny z konfiguracją na poziomie źródła danych
- **READ_UNCOMMITTED** - pozwala na odczyt danych zmienionych w ramach innych transakcji, ale jeszcze nie zatwierdzonych. Może powodować odczyty brudne, fantomowe i niepowtarzalne
- **READ_COMMITTED** - pozwala na odczyt danych zmienionych w ramach innych, zatwierdzonych już transakcji. Może powodować odczyty fantomowe i niepowtarzalne
- **REPEATABLE_READ** - zapewnia ochronę przed odczytami niepowtarzalnymi jednak nie chroni przed wystąpieniem fantomów
- **SERIALIZABLE** - zapewnia pełną ochronę (pełne wsparcie ACID) i najwyższy poziom izolacji kosztem małej wydajności

- **MANDATORY** - metoda musi działać w ramach transakcji, w przypadku jej braku wyrzucony zostaje wyjątek
- **NESTED** - jeśli transakcja istnieje metoda będzie działać w transakcji zagnieżdżonej, która może być niezależnie zatwierdzana/wycofywana
- **NEVER** - metoda nie może być wywoływana w ramach transakcji, jeśli tak się stanie dojdzie do wyjątku
- **NOT_SUPPORTED** - metoda nie może być wywoływana w ramach transakcji, jeśli tak się stanie bieżąca transakcja zostanie zawieszona
- **REQUIRED** - metoda musi działać w ramach transakcji, jeśli jej nie ma jest ona tworzona, jeśli jest wykonanie następuje w jej kontekście
- **REQUIRES_NEW** - zawsze tworzona jest nowa transakcja, jeśli transakcja już istnieje zostaje zawieszona
- **SUPPORTS** - metoda nie wymaga transakcji, jeśli transakcja już istnieje metoda będzie wykonywana w jej kontekście

- Preferowany sposób obsługi transakcji
- Konfiguracja może odbywać się przez pliki XML lub adnotacje

- Konfiguracja transakcji odbywa się przez znacznik `<tx:advice>` oraz znaczniki zagnieżdżone `<tx:attributes>` i `<tx:method>`
- Parametry transakcji dla wskazanej metody określone są przy użyciu atrybutów `isolation`, `propagation`, `read-only`, `rollback-for`, `no-rollback-for` i `timeout` znacznika `<tx:method>`
- Domyślnie zakładane jest istnienie menadżera transakcji o id równym "transactionManager" (jeśli jego nazwa jest inna należy ją zdefiniować przy użyciu atrybutu `transaction-manager`)

```
<tx:advice id="txAdvice" transaction-manager="txManager" >
  <tx:attributes>
    <tx:method name="add*" propagation="REQUIRED" />
    <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
  </tx:attributes>
</tx:advice>
```

- W celu przypisania aspektu realizującego obsługę transakcji konkretnym metodom aplikacji stosuje się znacznik `<aop:advisor>`

```
<aop:config>  
  <aop:advisor pointcut="execution(* *..Szkolenie.*(..))" advice-ref="txAdvice"/>  
</aop:config>
```

- Włączenie obsługi transakcji przy użyciu adnotacji odbywa się przez zadeklarowanie elementu **annotation-driven**
- Podobnie jak w przypadku XML, wskazanie id beana pełniącego rolę menadżera transakcji jest konieczne jeśli jego nazwa jest różna od "transactionManager"
- Adnotacja może być stosowana na poziomie klasy i/lub wybranych metod publicznych, posiada takie same atrybuty jak znacznik **<tx:method>**

```
<tx:annotation-driven transaction-manager="txManager" />
```

```
@Service
public class BankTransferService {
    @Transactional(propagation=Propagation.REQUIRES_NEW)
    public void doTransfer() {
        //...
    }
}
```

- Mapowanie obiektowo-relacyjne (ang. object-to-relational mapping) to zautomatyzowany proces zapewniania trwałości obiektów przy użyciu relacyjnej bazy danych
- ORM przekształca dane z modelu obiektowego do modelu relacyjnego i vice-versa
- ORM nie wymaga od programisty pisania tradycyjnego kodu SQL
- W skład frameworków ORM wchodzi:
 - API obsługujące podstawowe operacje CRUD
 - Język lub API służące do wykonywania bardziej skomplikowanych zapytań
 - Narzędzie pozwalające na definiowanie metadanych
 - „Maszyneria” zapewniająca obsługę pamięci podręcznej, leniwe ładowanie danych, propagację kaskadową itd.

- Spring nie implementuje własnego mechanizmu ORM, ale zapewnia integrację z najpopularniejszymi rozwiązaniami takimi jak Hibernate, JPA, Toplink, iBatis
- Ze swojej strony zapewnia:
 - Zintegrowany mechanizm obsługi transakcji
 - Zarządzanie i obsługę wyjątków
 - Zarządzanie niezbędnymi zasobami

60

Hibernate

Spring framework

- Najpopularniejszy i najbardziej rozbudowany framework ORM
- Zapoczątkowany przez Gavina Kinga
- Od roku 2003 rozwijany przez JBoss Inc.

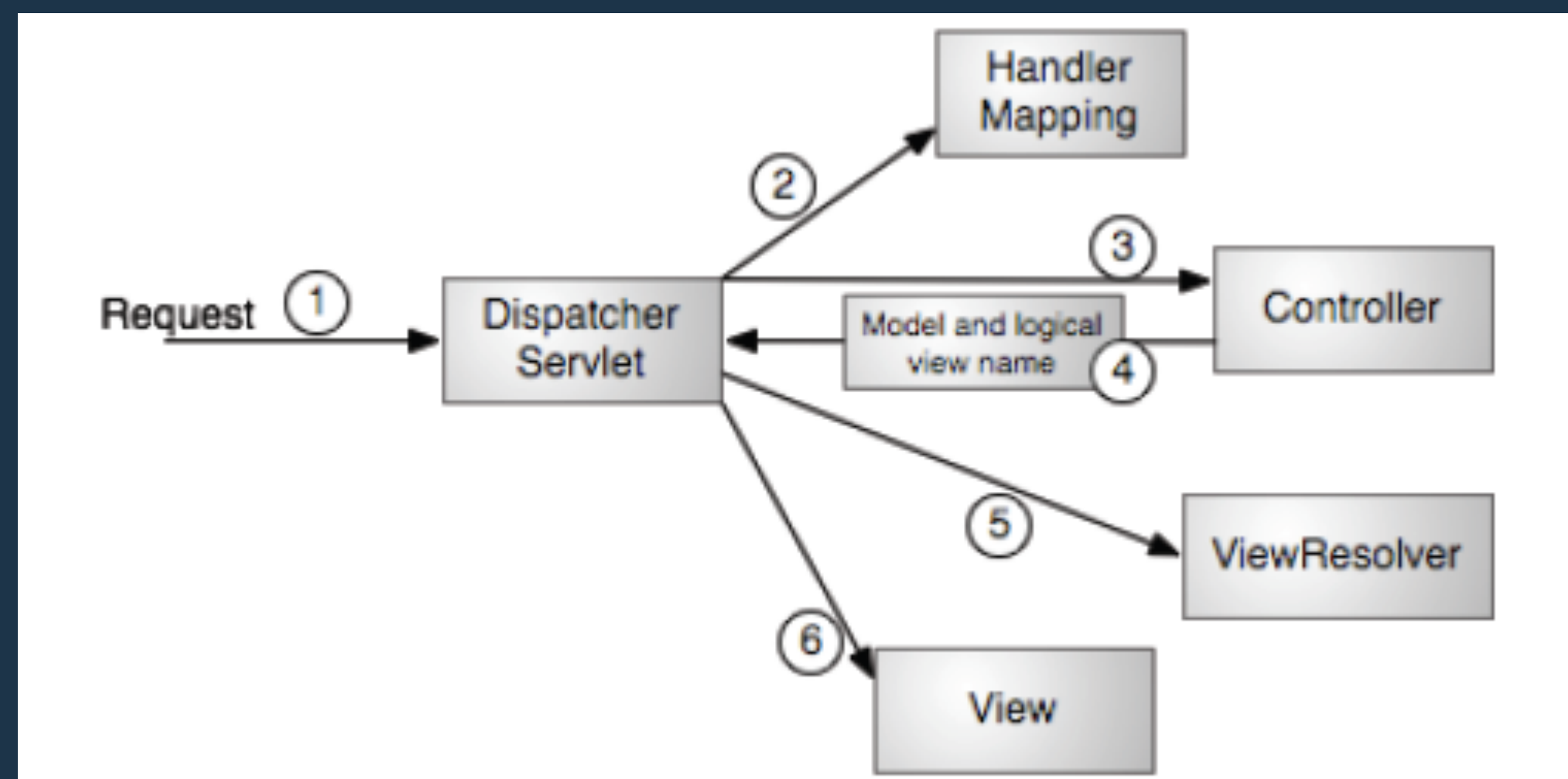
- Interfejs `org.hibernate.Session` jest używany do komunikacji z frameworkiem
- Obsługuje operacje zapisu i odczytu stanu obiektów trwałych
- Sesję uzyskujemy przy pomocy fabryki `org.hibernate.SessionFactory`
- Po zakończeniu operacji na obiektach trwałych sesję należy zamknąć
- W sesji znajduje się m.in. połączenie do bazy danych i pamięć podręczna dla obiektów trwałych

- Obiekty trwałe to obiekty, które mogą zostać odtworzone po zatrzymaniu i ponownym uruchomieniu aplikacji
- Każdy obiekt trwały może znajdować się w jednym z trzech stanów względem sesji
 - Przechodni (ang. transient) - instancja nie jest związana z sesją, nie posiada wartości klucza głównego
 - Trwały (ang. persistent) - instancja jest związana z sesją, posiada przypisaną wartość klucza głównego, może być reprezentowana przez rekord w bazie danych
 - Rozłączony (ang. detached) - instancja nie jest związana z sesją, posiada przypisaną wartość klucza głównego, może być reprezentowana przez rekord w bazie danych

- Framework webowy rozwijany w ramach niezależnego projektu Spring
- Stanowi implementację wzorca Model-View-Controller
- Zapewnia walidację i automatyczne mapowanie danych formularzy na obiekty
- Wspiera użycie wielu technologii widoku
- Pozwala obsługiwać żądania rozciągające się na kilka stron
- Umożliwia internacjonalizację aplikacji

- Implementacja wzorca MVC polega na logicznym wydzieleniu w aplikacji następujących warstw:
 - Warstwa modelu - przechowuje stan aplikacji, reprezentuje struktury danych prezentowane użytkownikom
 - Warstwa widoku - prezentuje dane przechowywane w modelu, pozwala na interakcję z systemem i powiadamia kontroler o podjętych akcjach użytkownika
 - Warstwa kontrolera - definiuje zachowanie aplikacji, przekształca żądania użytkownika na akcje wykonywane na modelu, decyduje o użyciu konkretnego widoku
- Cechy wzorca MVC:
 - Zapewnia swobodę w sposobie prezentowania danych
 - Centralizuje kontrolę przepływu sterowania w aplikacji
 - Separuje komponenty odpowiedzialne za przechowywanie, pozyskiwanie i zapis danych od komponentów widoku

- Za każdym razem kiedy użytkownik klika wybrany link albo zatwierdza formularz na serwer wysłane zostaje żądanie HTTP (ang. HTTP request)
- Poniżej przedstawiony został schemat obsługi żądania po stronie serwera



- Głównym elementem każdej aplikacji opartej o Spring MVC jest `DispatcherServlet` pełniący rolę front controller'a
- Jego konfiguracja odbywa się tradycyjnie w pliku `web.xml`
- Nazwa servletu ma istotne znaczenie - podczas ładowania servletu następuje próba wczytania kontekstu Spring z pliku `<nazwa>-servlet.xml`

```
<servlet>
  <servlet-name>bank</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>bank</servlet-name>
  <url-pattern>*.mvc</url-pattern>
</servlet-mapping>
```