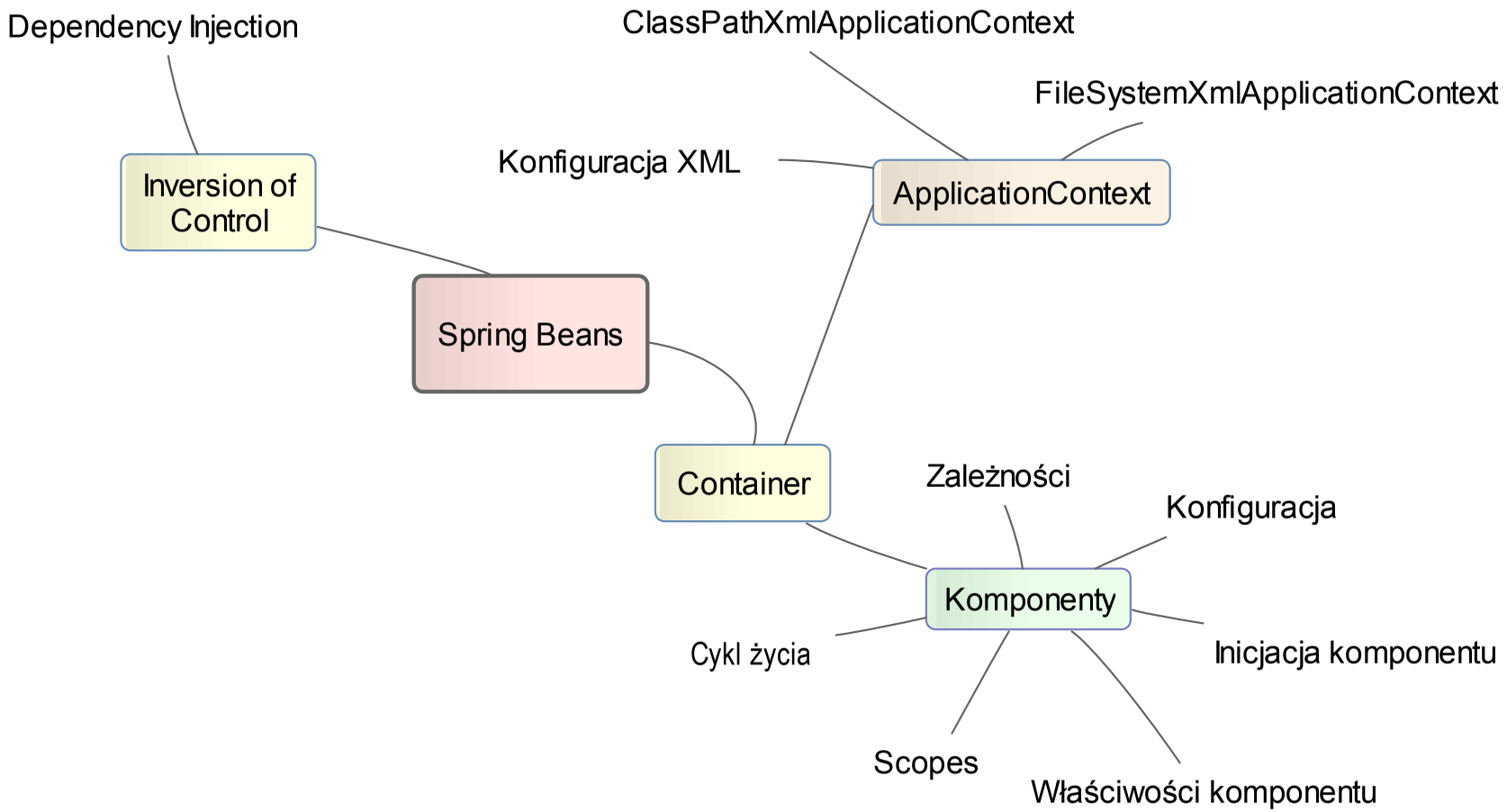
JAVATECH

Spring Beans

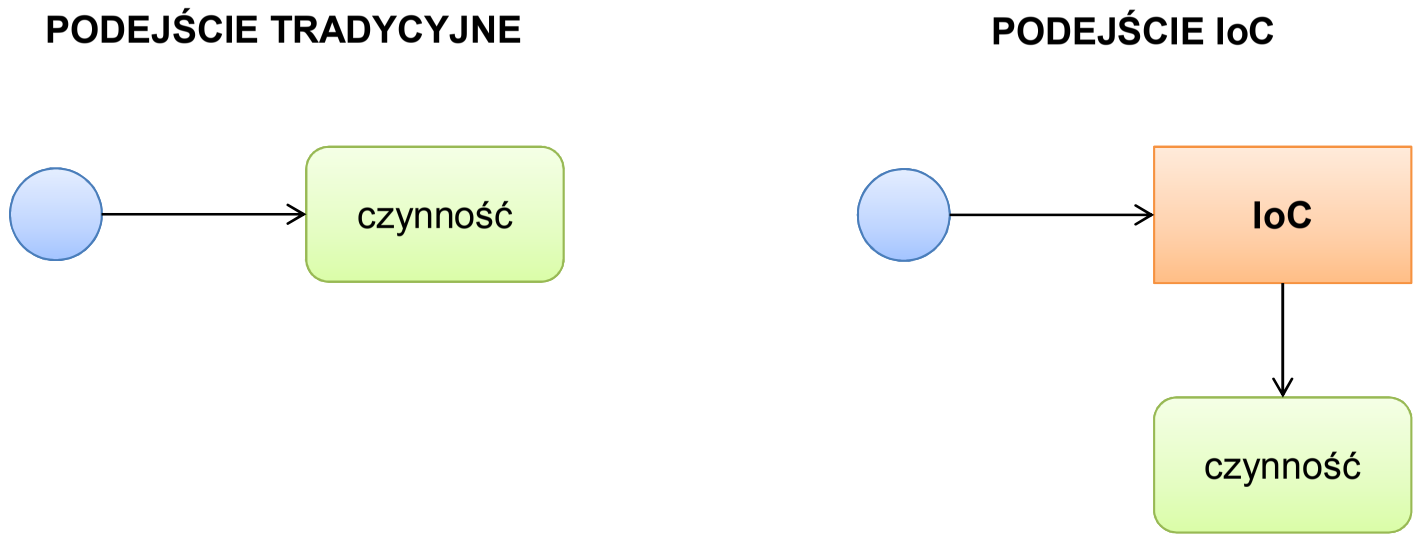
Realizacja koncepcji Inversion of Control
za pomocą Spring Container

Plan


Dependency Injection
Inversion of Control
Spring Beans
ClassPathXmlApplicationContext
FileSystemXmlApplicationContext
Konfiguracja XML
ApplicationContext
Container
Zależności
Konfiguracja
Komponenty
Cykl życia
Inicjacja komponentu
Scopes
Właściwości komponentu

Inversion of Control

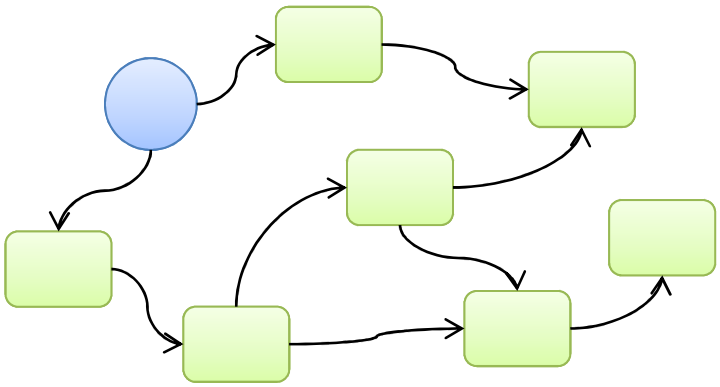
- wzorzec architektoniczny polegający na przeniesieniu na zewnątrz komponentu (np. obiektu) odpowiedzialności za kontrolę wybranych czynności



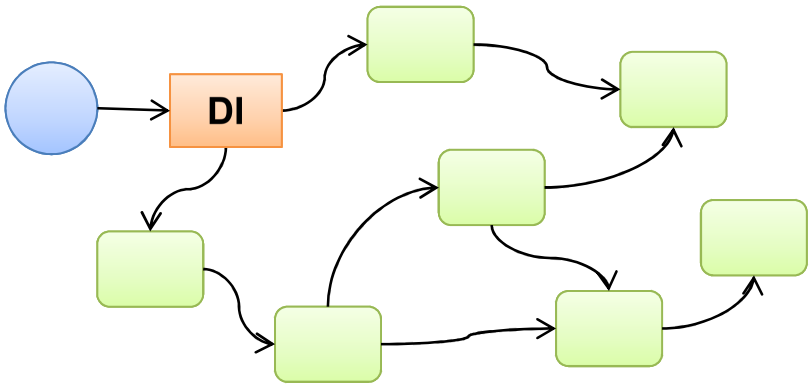
Dependency Injection

- wzorzec architektoniczny polegający na usunięciu bezpośrednich zależności pomiędzy komponentami systemu
- odpowiedzialność za tworzenie obiektów przeniesione zostaje do zewnętrznej fabryki obiektów – kontenera
- kontener na żądanie tworzy obiekt bądź zwraca istniejący z puli ustawiając powiązania z innymi obiektami

PODEJŚCIE TRADYCYJNE

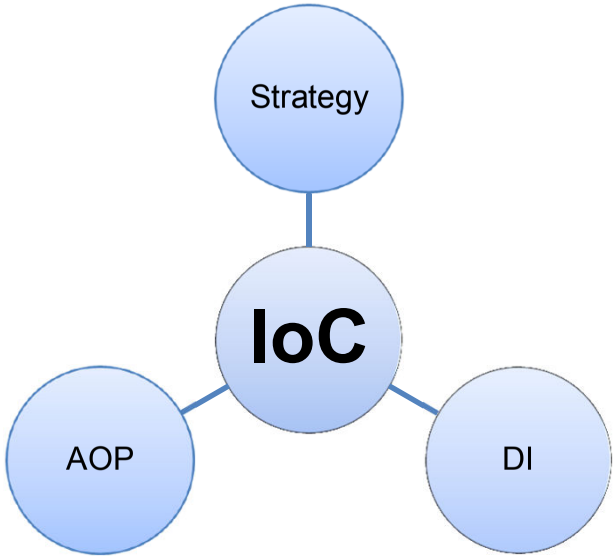


PODEJŚCIE DI

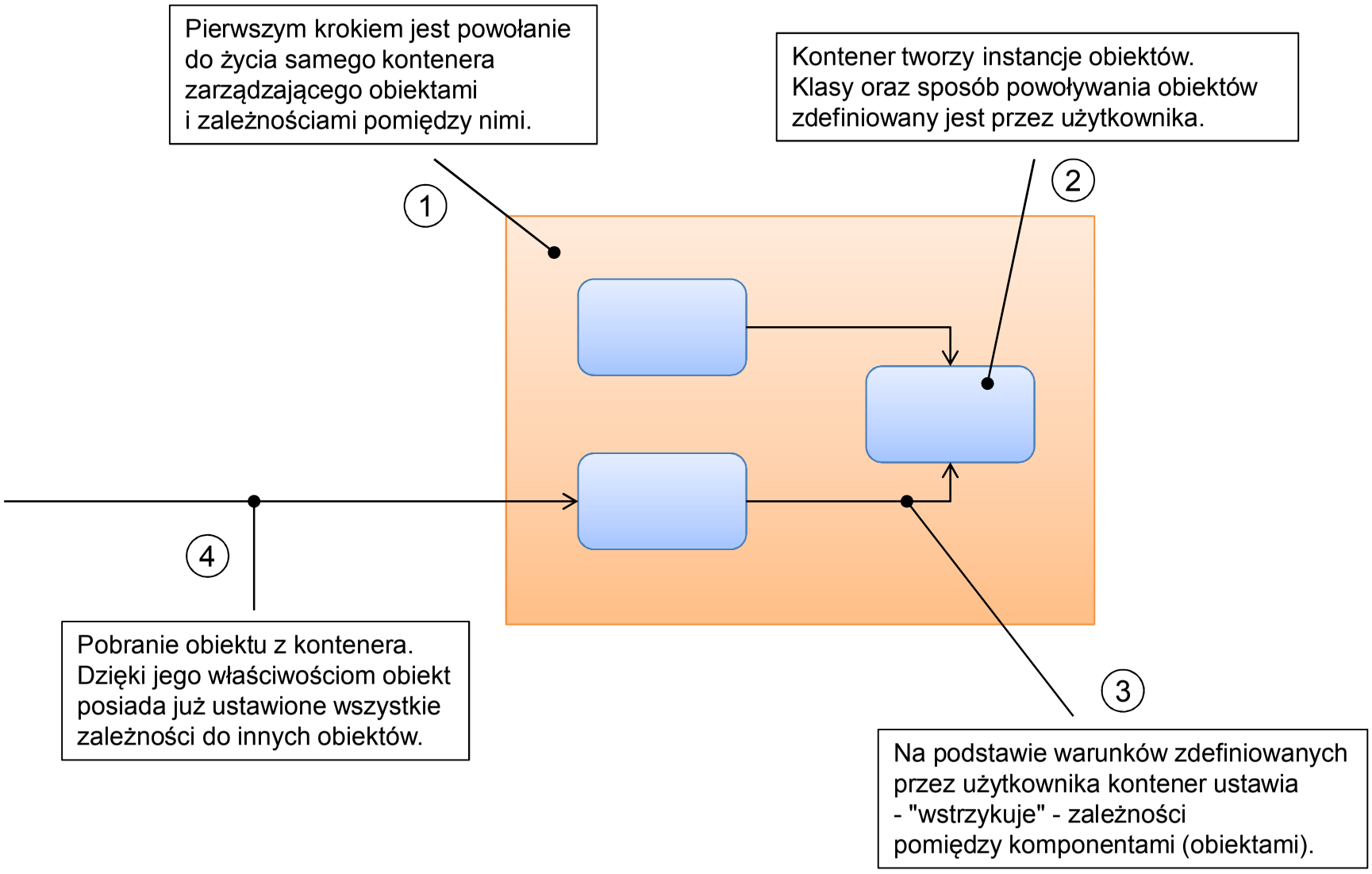


IoC != DI

- Inversion of Control często błędnie utożsamiane jest jednoznacznie z Dependency Injection.
- Dependency Injection to szczególny przypadek IoC, który obejmuje szerszy krąg przypadków.



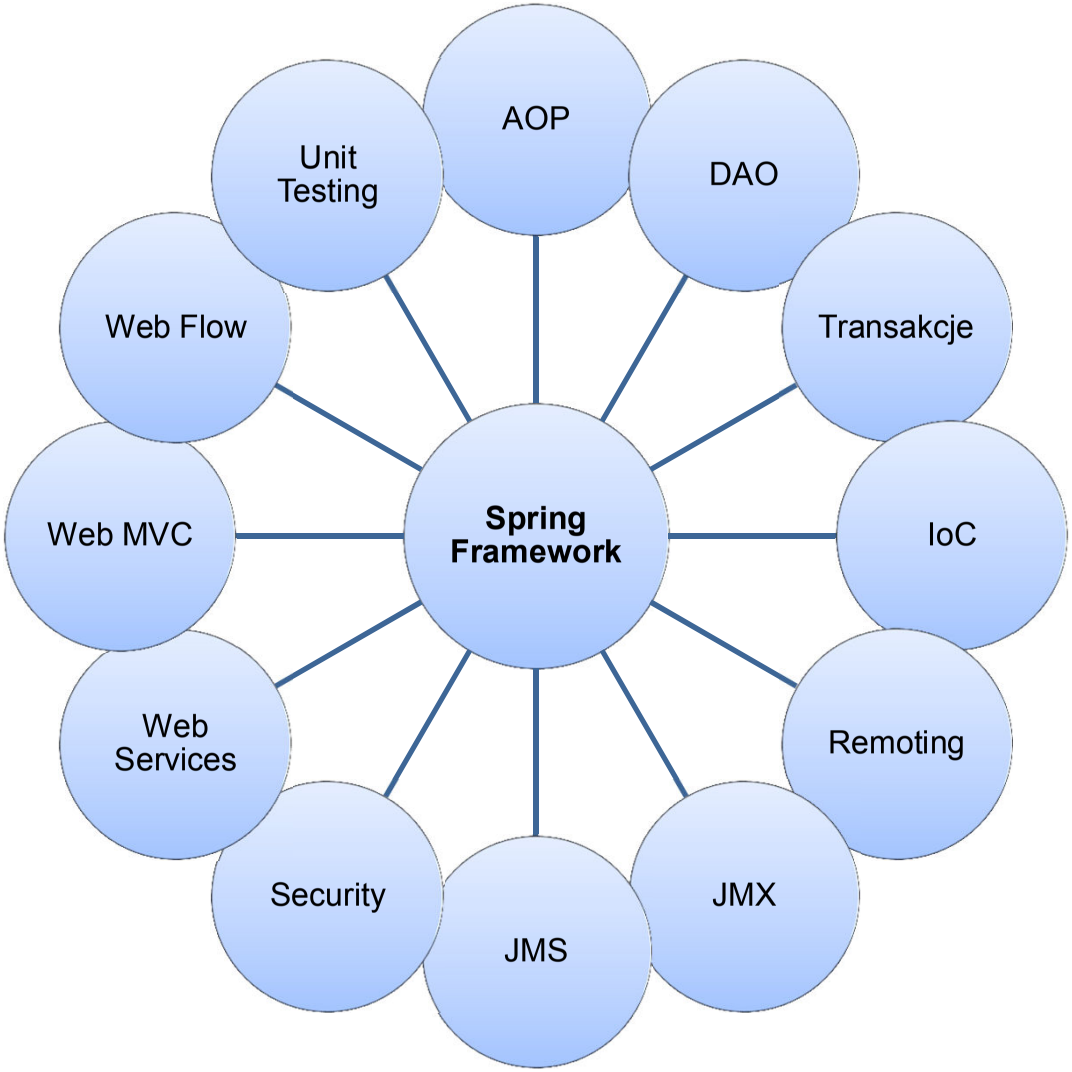
Działanie kontenera DI



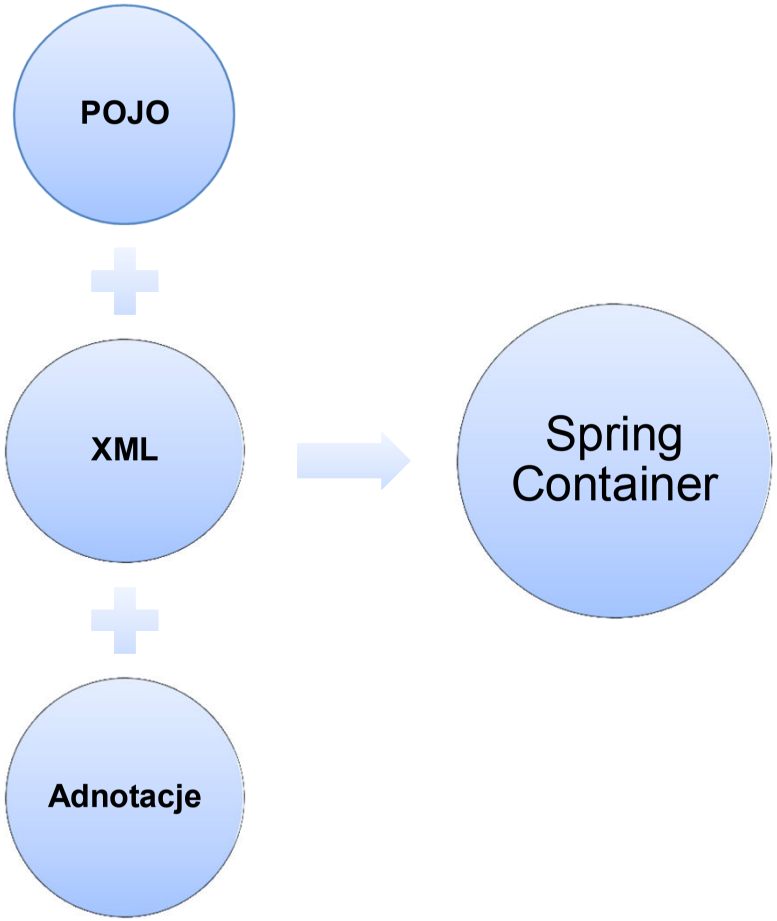
Spring jako alternatywa dla Java EE

- Komponenty EJB (2.1)
  - "ciężkie" i skomplikowane
  - trudne do testowania
- Spring (2003)
  - lekki
  - prosty w konfiguracji
  - łatwy do testowania
  - duża funkcjonalność dodatkowa

Spring Framework



Spring jako kontener

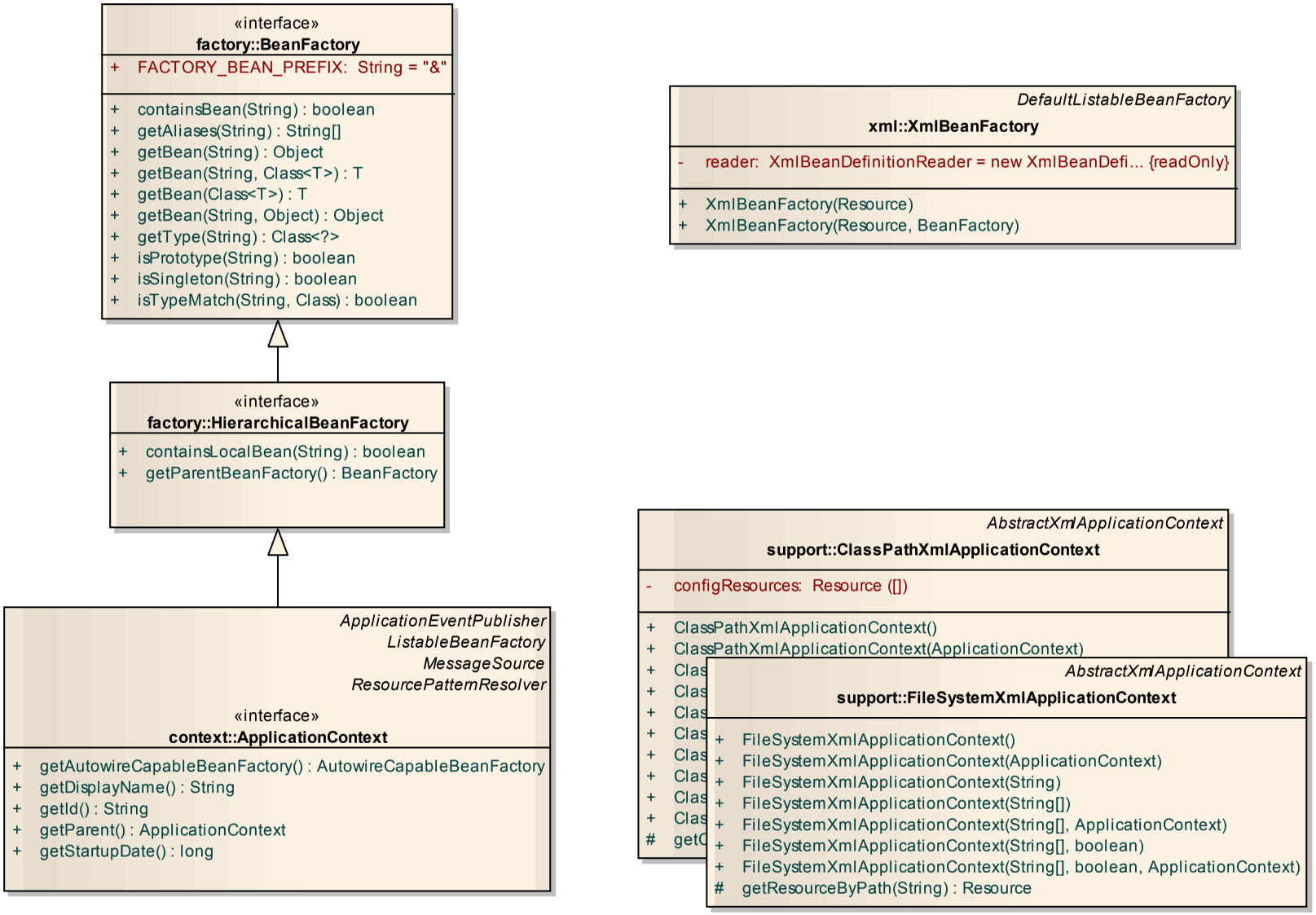


JAVATECH

Podstawowe pakiety

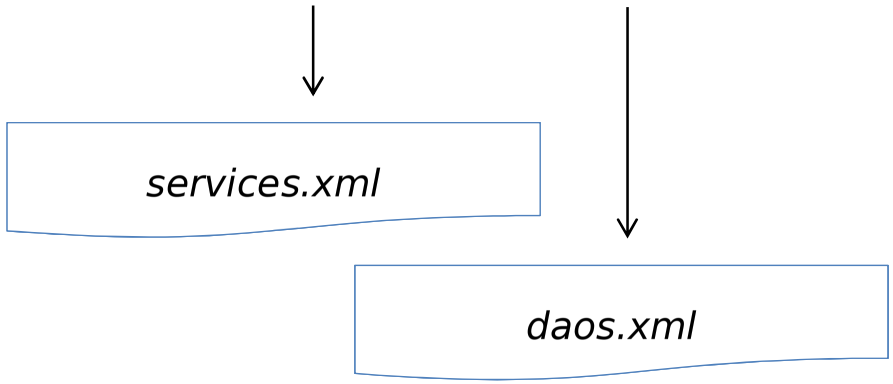
- Konstrukcja springowego IoC osadzona jest w dwóch pakietach
  - org.springframework.beans
  - org.springframework.context

Podstawowe interfejsy



Inicjalizacja kontenera

```
ApplicationContext context =
        new ClassPathXmlApplicationContext(
                new String[] {"services.xml", "daos.xml"});
```



Inicjalizacja kontenera polega na wywołaniu konstruktora wybranej implementacji wraz z odpowiednimi argumentami. Argumentami będą lokalizacje pliku (plików) konfiguracyjnych w kontekście classpath lub filesystem (w zależności od wybranej implementacji.

Plik konfiguracyjny XML

```
<?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.xsd">

<bean id="..." class="...">
</bean>

<bean id="..." class="...">
</bean>

</beans>
```

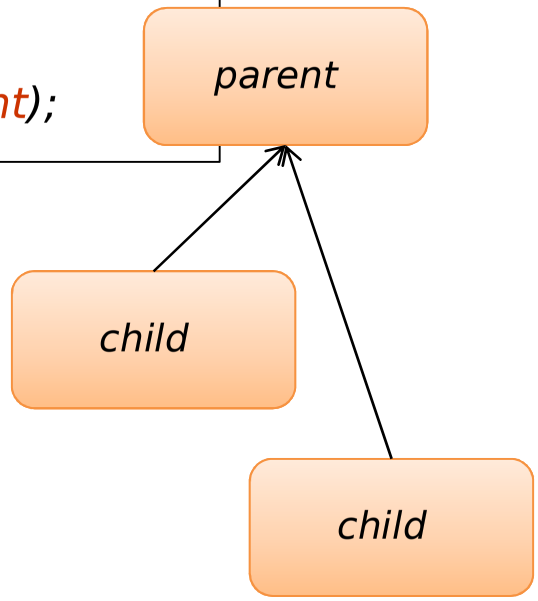
Złożony plik konfiguracyjny

```
<?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.xsd">

<import resource="services.xml"/>
<import resource="resources/messageSource.xml"/>
<import resource="/resources/themeSource.xml"/>
<import resource="classpath:/META-INF/spring/dao.xml"/>

</beans>
```

Złożony plik konfiguracyjny daje szerokie możliwości konfiguracji systemu. Pozwala utrzymać większy porządek dzięki podziałowi na mniejsze części, umożliwia wprowadzenie dynamicznego zestawu komponentów w zależności od zawartych np. w classpath bibliotek.

Parent/Child

```
XmlBeanFactory parent =
  new XmlBeanFactory(
    new ClassPathResource("/applicationContext.xml"));

XmlBeanFactory child =
  new XmlBeanFactory(
    new ClassPathResource("/services.xml"), parent);
```



Stworzenie kilku kontenerów oraz stworzenia z nich hierarchii pozwala na lepsze zarządzanie istniejącymi obiektami i zasobami.
Komponenty z nadrzędnych kontenerów nie "widzą" komponentów z podrzędnych jednak odwrotna relacja istnieje. Dzięki temu można wyodrębnić część wspólną komponentów widoczną dla kontenerów podrzędnych.
Taką architekturę wykorzystuje się np. w Spring MVC.

Definicja komponentu

- Identyfikator komponentu
- Nazwa klasy komponentu
- Właściwości
- Zależności
- Inicjalizacja komponentu
- Tryby inicjalizacji i pracy komponentu

Identyfikator komponentu

- Nazwa komponentu musi być unikalna w obrębie pojedynczego kontenera, w obrębie którego komponent działa.
- Nazwę komponentu definiuje atrybut "id" lub "name" znacznika "bean". W przypadku wykorzystania atrybutu "name" możliwe jest zdefiniowanie wielu nazw oddzielonych od siebie (,) lub (;) ewentualnie spacją.
- Nazwa nie jest obowiązkowa. Cecha ta wykorzystywana jest w komponentach zagnieżdżonych, technicznych oraz wykorzystując mechanizm autowire.

```
<bean id="..." name="..." >
</bean>
```

Identyfikator komponentu (cd)

```
<bean id="personController">
</bean>

<bean id="securityService">
</bean>
```

Warto nadawać komponentom jednoznaczne nazwy
o jednorodnej strukturze, pozwala to na łatwiejszą integrację
z innymi usługami Spring np. elementami proxy, transakcjami,
AOP itp.

Aliasy

```
<alias name="dataSourceA" alias="dataSourceB"/>

<alias name="dataSourceA" alias="myDataSource" />
```

Istnieje możliwość nadania komponentom dodatkowej nazwy poza ich definicją. Dzięki temu można wprowadzić aliasy w kontenerach podrzędnych, w innych plikach konfiguracyjnych itp.

Inicjacja komponentu

- za pomocą konstruktora
- za pomocą statycznej metody fabrykującej
- za pomocą metody fabrykującej obiektu

Bezparametrowy konstruktor

```
<bean id="exampleBean"
        class="examples.ExampleBean"/>

<bean name="anotherExample"
        class="examples.SecondExampleBean"/>
```

Konstruktor z parametrami

```
<bean id="exampleBean" class="examples.ExampleBean">
        <constructor-arg value="1"/>
        <constructor-arg value="2"/>
        <constructor-arg value="3"/>
</bean>
```

Konstruktor z parametrami (cd)

```
<bean class="example.BeanTest">
        <constructor-arg type="java.lang.Integer" value="10" />
        <constructor-arg type="java.lang.String" value="10" />
</bean>


<bean class="example.BeanTest">
        <constructor-arg index="0" value="10" />
        <constructor-arg index="1" value="10" />
</bean>
```

Statyczna metoda fabrykująca

```
<bean id="exampleBean"
      class="examples.ExampleBeanFactory"
                factory-method="createInstance"/>
```

W tym przypadku argument "class" nie oznacza klasy obiektu tworzonego przez kontener a klasę zawierającą określoną statyczną metodę.

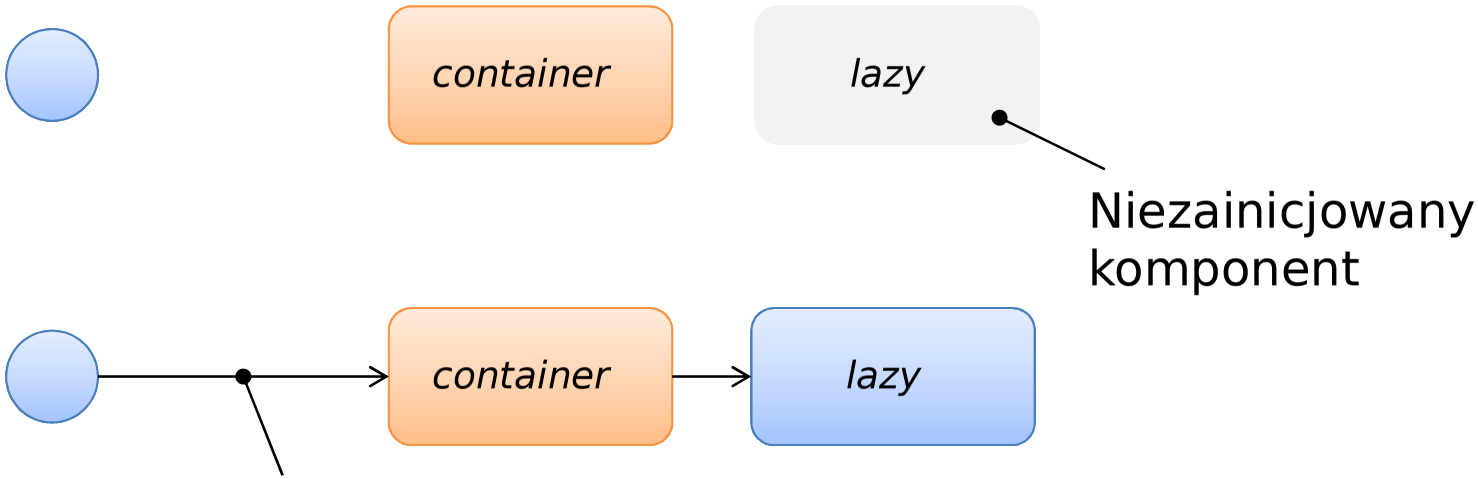
Metoda fabrykująca obiektu

```
<bean id="myFactoryBean"
      class="example.ExampleBeanFactory">
...
</bean>

<bean id="exampleBean"
      factory-bean="myFactoryBean"
                factory-method="createInstance"/>
```

W tym przypadku nie używany jest atrybut "class".

lazy-init

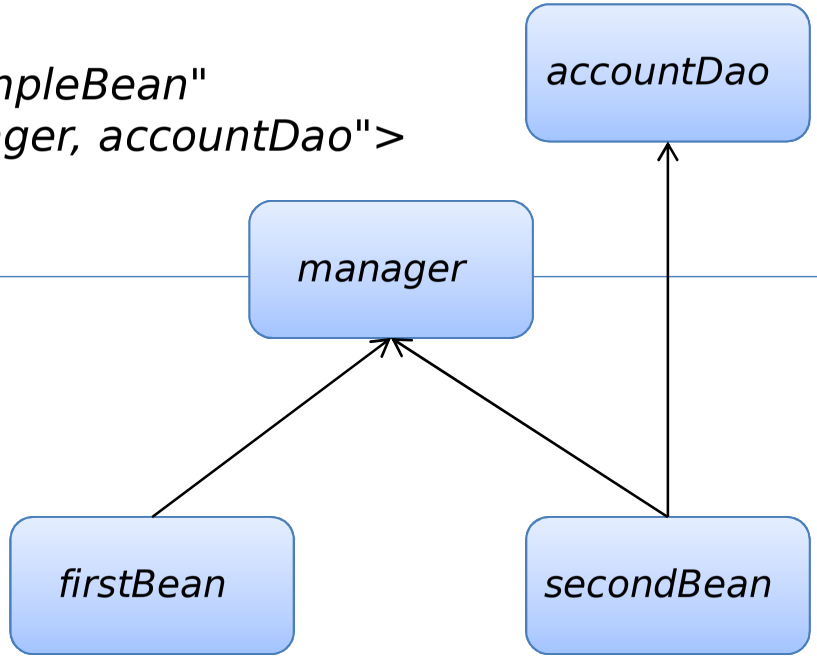
```
<bean id="lazy"
      class="lab.spring.ExpensiveToCreateBean"
      lazy-init="true"/>
```



Użycie leniwej inicjalizacji komponentu powoduje, że zostanie on utworzony dopiero przy pierwszym odwołaniu do niego. Pozwala to na regulację użycia zasobów i przyspiesza inicjalizację kontenera.

depends-on

```
<bean id="manager" class="lab.spring.ManagerBean" />

<bean id="accountDao" class="lab.spring.JdbcAccountDao" />

<bean id="firstBean"
      class="lab.spring.FirstExampleBean"
            depends-on="manager"/>

<bean id="secondBean"
      class="lab.spring.SecondExampleBean"
            depends-on="manager, accountDao">
</bean>
```



Dziedziczenie definicji

```
<bean id="inheritedTestBean" abstract="true"
      class="org.springframework.beans.TestBean">
      <property name="name" value="parent"/>
      <property name="age" value="1"/>
</bean>


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

Właściwości

```
public class org.apache.commons.dbcp.BasicDataSource
        implements javax.sql.DataSource {

    protected java.lang.String driverClassName;
    protected java.lang.String password;
    protected java.lang.String url;
    protected java.lang.String username;

    public synchronized java.lang.String getDriverClassName(){
      ...
    }

    public synchronized void setDriverClassName(java.lang.String
    driverClassName) {
      ...
    }
}
```

Konfiguracja właściwości

```
<bean id="myDataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
      <property name="driverClassName">
            <value>com.mysql.jdbc.Driver</value>
      </property>
      <property name="url">
            <value>jdbc:mysql://localhost:3306/mydb</value>
      </property>
      <property name="username">
            <value>root</value>
      </property>
</bean>
```

Podanie w konfiguracji właściwości elementu *value* powoduje wywołanie odpowiedniego settera po utworzeniu komponentu w kontenerze. Przykładowo dla własciwości *driverClassName* zostanie wywołana metoda *setDriverClassName.*

Referencje

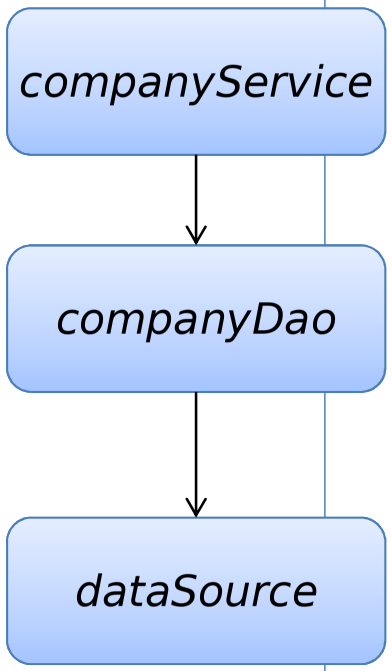
```
public class CompanyServiceImpl
        implements CompanyService {

    private CompanyDao companyDao;

    public void setCompanyDao(CompanyDao companyDao) {
        this.companyDao = companyDao;
    }
}
```

```
public class CompanyDao Impl
        implements CompanyDao{

        private DataSource dataSource;

        public void setDataSource(DataSource dataSource) {
                this.dataSource = dataSource;
        }

}
```

Klasyczny zestaw klas w aplikacji wielowarstwowej, warstwa usług udostępniająca funkcjonalność, dao dla dostępu do danych |
i datasource definiujący źródło danych – zazwyczaj bazę danych.

Konfiguracja Spring

```
<bean id="companyService" class="spring.CompanyServiceImpl">
        <property name="companyDao" ref="companyDao"/>
</bean>

<bean id="companyDao" class="dao.CompanyDaoImpl">
        <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName">
       <value>com.mysql.jdbc.Driver</value>
  </property>
  <property name="url">
      <value>jdbc:mysql://localhost:3306/mydb</value>
  </property>
  <property name="username">
      <value>root</value>
  </property>
</bean>
```



Kolekcje

```
<property name="someList">
      <list>
        <value>red</value>
        <value>blue</value>
        <value>green</value>
      </list>
</property>

<property name="someSet">
      <set>
        <value>red</value>
        <value>red</value>
        <value>green</value>
      </set>
</property>
```

W konfiguracji spring istnieje możliwość zdefiniowania kolekcji jako wartości przekazanej do właściwości komponentu.

Mapy i properties

```
<property name="someMap">
    <map>
        <entry key="item1" value="To jest tekst"/>
        <entry key ="item2" value-ref="dataSource"/>
    </map>
</property>
```

```
<property name="someProperties">
    <props>
    <prop key="administrator">administrator@example.org</prop>
    <prop key="support">support@example.org</prop>
    <prop key="development">development@example.org</prop>
    </props>
</property>
```

Wartości null

```
<bean class="ExampleBean">
  <property name="email"><value></value></property>
</bean>

<bean class="ExampleBean">
  <property name="email"><null/></property>
</bean>
```

Przekazanie wartości null do właściwości komponentu wymaga użycia specjalnego elementu <null/>

JAVATECH

Obsługa cyklu życia komponentu

- W ramach kontenera możemy zadeklarować obsługę życia komponentu.
- Istnieje możliwość przechwycenia momentu utworzenia komponentu jak i jego usunięcia.

Utworzenie komponentów

```
<bean id="personService"
      class="lab.spring.PersonServiceImpl"
            init-method="init" />
```

```
public class PersonServiceImpl
      implements PersonService, InitializingBean {

  public void afterPropertiesSet()
      throws Exception {

  }

}
```

Usunięcie komponentów

```
<bean id="personService"
      class="lab.spring.PersonServiceImpl"
            destroy-method="destroy"/>
```

```
public class PersonServiceImpl
      implements PersonService, DisposableBean {

  public void destroy() throws Exception {

  }

}
```

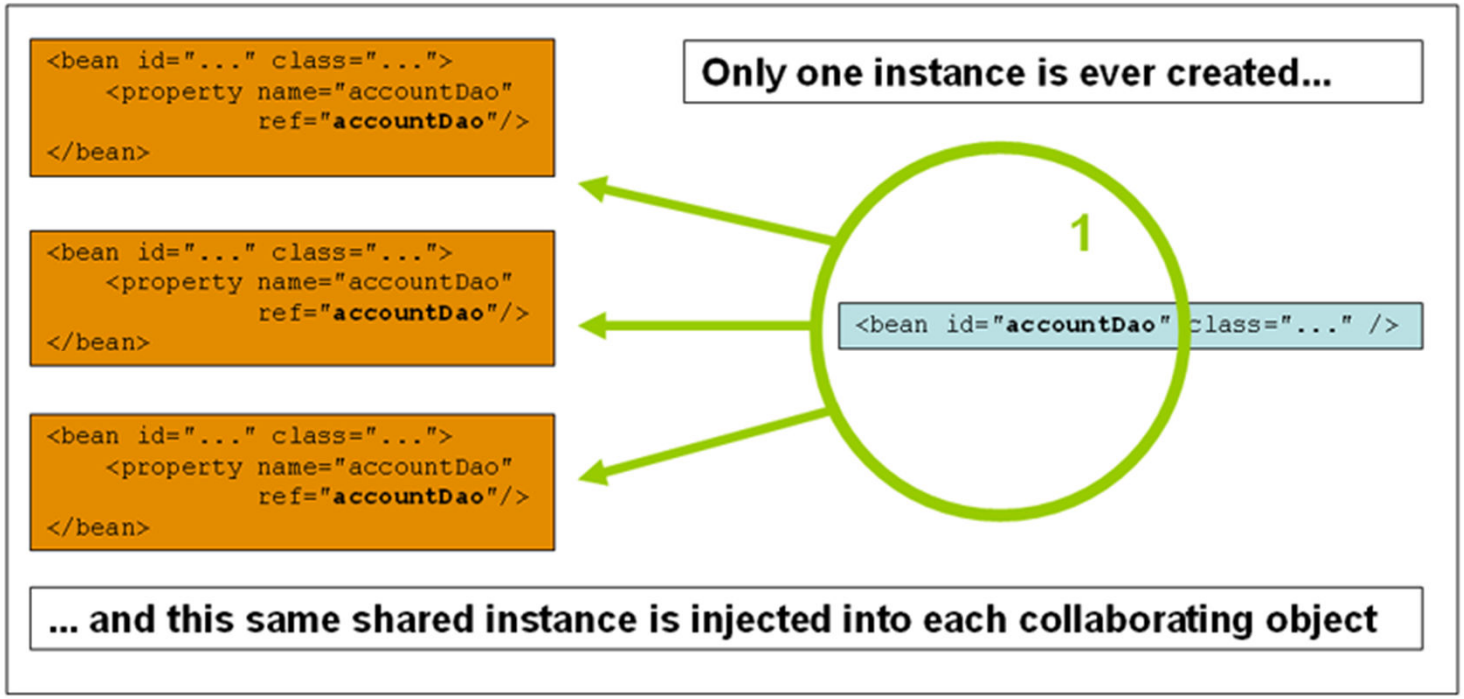
ShutdownHook

```
public class ShutdownHook implements Runnable {
  private ConfigurableListableBeanFactory beanFactory;
  public ShutdownHook(ConfigurableListableBeanFactory beanFactory) {
    this.beanFactory = beanFactory;
  }
  public void run() {
    this.beanFactory.destroySingletons();
  }
}

public class ShutdownHookDemo {
  public static void main(String[] args) throws IOException {
    XmlBeanFactory factory = new XmlBeanFactory(
    new ClassPathResource("/applicationContext.xml"));
    Runtime.getRuntime().addShutdownHook(new Thread(new
        ShutdownHook(factory)));
    ...
    ...
  }
}
```

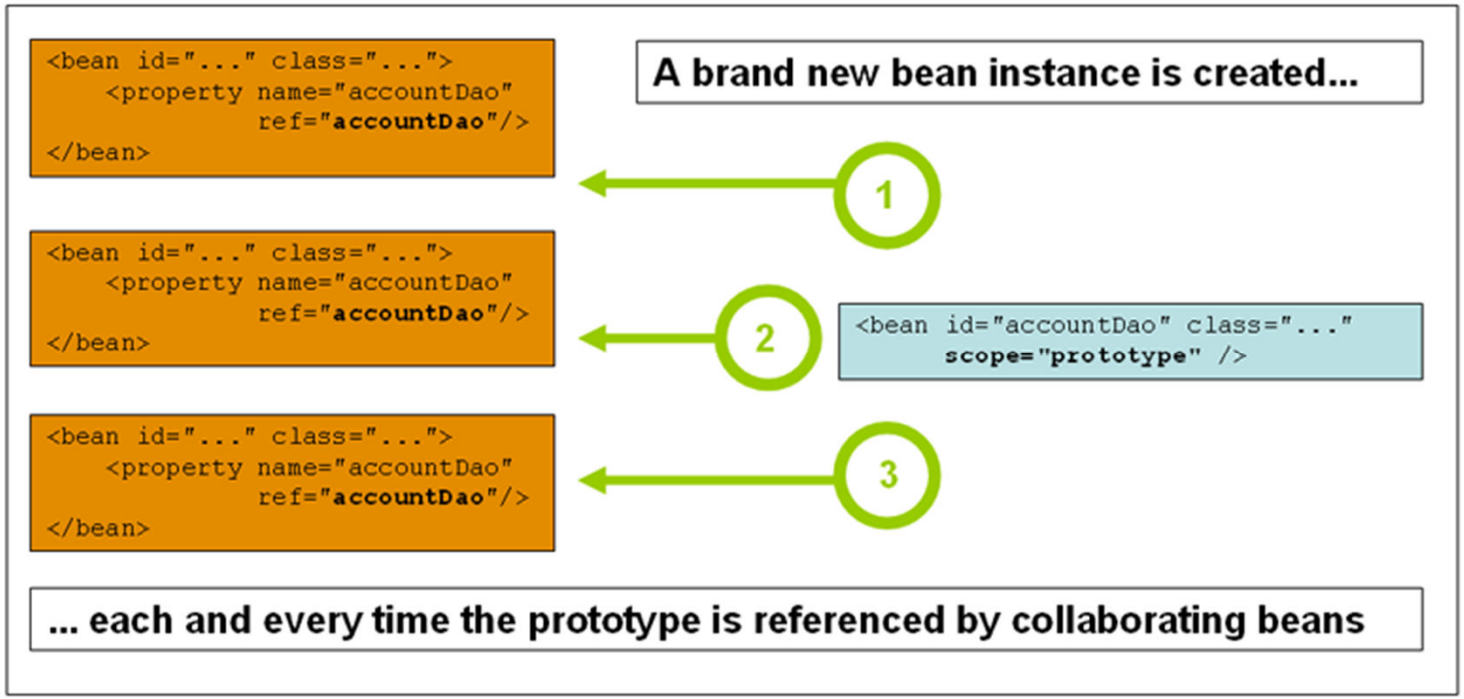
Bean scopes

- singleton
- prototype
- session
- request
- global session

Singleton


<bean id="..." class="...">
<property name="accountDao"
ref="accountDao"/>
</bean>
<bean id="..." class="...">
<property name="accountDao"
ref="accountDao"/>
</bean>
<bean id="..." class="...">
<property name="accountDao"
ref="accountDao"/>
</bean>
Only one instance is ever created...
1
<bean id="accountDao" class="..." />
... and this same shared instance is injected into each collaborating object

Prototype

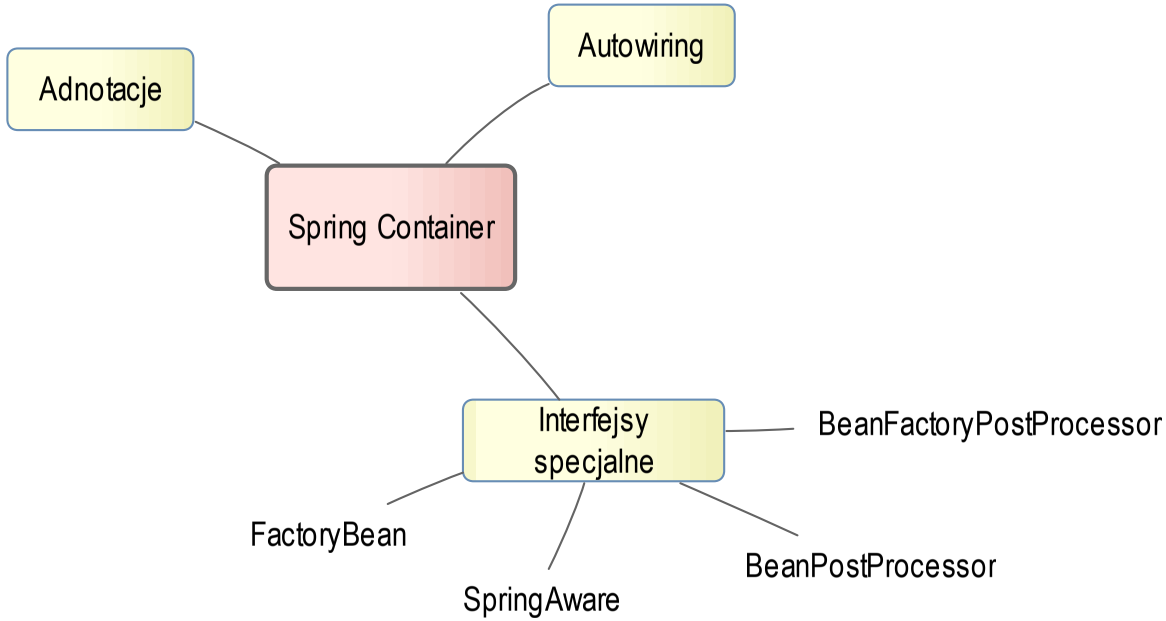

A brand new bean instance is created...
<bean id="..." class="...">
<property name="accountDao"
ref="accountDao"/>
</bean>
1
<bean id="..." class="...">
<property name="accountDao"
ref="accountDao"/>
</bean>
2
<bean id="accountDao" class="..."
scope="prototype" />
<bean id="..." class="...">
<property name="accountDao"
ref="accountDao"/>
</bean>
3
... each and every time the prototype is referenced by collaborating beans

JAVATECH

Spring Container

Zaawansowane funkcje kontenera komponentów
Spring Framework

Plan


Adnotacje
Autowiring
Spring Container
Interfejsy specjalne
BeanFactoryPostProcessor
FactoryBean
SpringAware
BeanPostProcessor

JAVATECH

Kontener Spring

- Kontener Spring – zaawansowana fabryka obiektów.
- Posiada szereg punktów i mechanizmów umożliwiających rozszerzanie jego możliwości bez konieczności dziedziczenia po ApplicationContext co zwiększa uniwersalność i elastyczność rozwiązania.
- Do tego celu został przygotowany zestaw interfejsów przykładowo
  - FactoryBean
  - BeanPostProcessor
  - BeanFactoryPostProcessor

FactoryBean

- Fabryka w fabryce.
- Specjalny rodzaj obiektu, który pozwala przeprowadzić własny sposób na osadzenie komponentu w kontenerze.

```
public interface FactoryBean {
    Object getObject() throws Exception;
    Class getObjectType();
    boolean isSingleton();
}
```

Własna fabryka komponentów to obiekt klasy implementującej interfejs FactoryBean

FactoryBean

```
public class PersonFactoryBean implements FactoryBean {

        public Object getObject() throws Exception {
                return new Person();
        }
        public Class getObjectType() {
                return Person.class;
        }
        public boolean isSingleton() {
                return false;
        }

}
```

Factory Bean (cd)

```
<bean id="person" class="lab.spring.PersonFactoryBean"/>
```

```
ApplicationContext ctx = new
ClassPathXmlApplicationContext("/applicationContext.xml");

Person p = (Person) ctx.getBean("person");

p = (Person) ctx.getBean("person");
p = (Person) ctx.getBean("person");
p = (Person) ctx.getBean("person");
```

W przypadku ustawienia komponentu jako singleton zawsze otrzymamy tą samą instancję, w przeciwnym wypadku za każdym razem wywołana zostanie metoda getObject()

JAVATECH

BeanPostProcessor

- Pozwala na przechwycenie procesu tworzenia poszczególnych beanów i np. do logowania procesu inicjalizacji komponentów czy nawet zmianę ich funkcjonalności
- Obiekt klasy implementującej interfejs BeanPostProcessor powoływany do życia jest przez IoC w początkowej fazie działania przed innymi komponentami.
- Jest specjalnie traktowany np. nie bierze udziału w procesie auto-proxying (AOP).
- W ramach kontekstu może być zdefiniowanych kilka beanów implementujących ten interfejs.
- W celu określenia kolejności przetwarzania należy zaimplementować interfejs Ordered.

BeanPostProcessor

```
public class MyBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
            throws BeansException {
            ...
            return bean;   //tu potencjalnie możemy zwrócić dowolny obiekt
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
            throws BeansException {
            ...
            return bean;
    }
}
```

JAVATECH

BeanFactoryPostProcessor

- Działa podobnie jak BeanPostProcessor jednak jest jedna zasadnicza różnica.
- BeanFactoryPostProcessor operuje na metadanych konfiguracji komponentu wobec tego pozwala na ich zmianę jeszcze przed jego zainicjowaniem.
- W ramach kontekstu może być zdefiniowanych kilka komponentów realizujących taką konfigurację.
- W celu określenia kolejności przetwarzania należy zaimplementować interfejs Ordered.

BeanFactoryPostProcessor

```
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {

    @Override
    public void postProcessBeanFactory(
                        ConfigurableListableBeanFactory beanFactory)
            throws BeansException {

    }

}
```

Wbudowane BeanFactoryPostProcessor

- AspectJWeavingEnabler
- CustomAutowireConfigurer
- CustomEditorConfigurer
- CustomScopeConfigurer
- PreferencesPlaceholderConfigurer
- PropertyOverrideConfigurer
- PropertyPlaceholderConfigurer
- ServletContextPropertyPlaceholderConfigurer

PropertyPlaceholderConfigurer

- Pozwala na użycie zewnętrznych plików do przechowywania wartości właściwości komponentu.
- Dzięki temu rozwiązaniu możliwe jest np. zmienienie konfiguracji komponentu bez konieczności przebudowywania archiwum aplikacji czy ingerowania
  w jej integralność.

Przykład

```
<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>
```

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

jdbc.properties

BeanFactoryAware

```
public class PersonServiceImpl
        implements PersonService, BeanFactoryAware {

        public void setBeanFactory(BeanFactory beanFactory)
                throws BeansException {

        }

}
```

BeanNameAware

```
public class PersonServiceImpl
        implements PersonService, BeanNameAware {

        public void setBeanName(String name) {

        }

}
```

JAVATECH

Wstrzykiwanie metod

- Nie wszystkie fasolki są singletonami
- Korzystanie z "niesingletonowej" fasolki w singletonie może wymagać wyszukania fasolki za pomocą getBean()...
- … ale to spowoduje zależność naszego kodu od Springa
- … i wymaga podania nazwy fasolki w kodzie
- … i nie jest to IoC ;)
- Wstrzykiwanie metod pozwala temu zaradzić...

JAVATECH

Wstrzykiwanie metod (2)

- Kontener może dostarczyć implementację zadanej metody w naszym kodzie.
- Implementowana metoda będzie zwracała fasolkę tak jak skonfigurowane w pliku XML
- Metoda musi mieć postać:
- <public|protected> [abstract] <zwracany-typ> nazwaMetody();

Wstrzykiwanie metod - przykład

```
<bean id="textProducer" class="lab.spring.TextProducer" scope="prototype">
  ...
</bean>

<bean id="hello" class="lab.spring.Hello">
  <lookup-method name="createTextProducer" bean="textProducer"/>
</bean>
```

```
public abstract class Hello {

   public void printHello {

            TextProducer textProducer = createTextProducer();
            System.out.println(textProducer.getText());
   }

            protected abstract TextProducer createTextProducer();
}
```

Autowiring

```
<bean id="personService" class="spring.PersonServiceImpl" autowire="byType"/>

<bean id="personDao" class="dao.PersonDao" autowire="byType"/>

<bean id="myDataSource"  class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
  </property>
  <property name="url">
        <value>jdbc:mysql://localhost:3306/mydb</value>
 </property>
<property name="username">
        <value>root</value>
</property>
</bean>
```

Automatyczne wiązanie może być zdefiniowane jako wiązanie:
byType
byName
contructor
autodetect

JAVATECH

Domyślne automatyczne wiązanie

- Istnieje możliwość zdefiniowania domyślnego automatycznego wiązania i jego rodzaju poprzez dodanie atrybutu do elementu *beans* w pliku konfiguracyjnym.

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

@Autowired

```
public class PersonDao{

        @Autowired
        private DataSource dataSource;

        public void setDataSource(DataSource dataSource) {
                this.dataSource = dataSource;
        }

}
```

```
<context:annotation-config />
```

W celu zdefiniowania automatycznego wiązania komponentów można użyć adnotacji. Wymaga to dodatkowej konfiguracji kontekstu Spring informującego o tym iż jego konfiguracja odbywa się w taki sposób.

JAVATECH

Nowe adnotacje w Spring 3.0

- adnotacje JavaConfig:
  - @Configuration
  - @Bean
  - @DependsOn
  - @Primary
  - @Lazy
  - @Import, @ImportResource
  - @Value
- adnotacje JSR 330 (Dependency Injection for Java):
  - @Inject, @Qualifier, @Named, @Provider

@Configuration i @Bean – klasa konfiguracji

- @Configuration określa, że ta klasa będzie zawierała metody dostarczające komponenty.
- @Bean rejestruje obiekt zwrócony przez metodę jako komponent o nazwie takiej jak metoda.
- Zależności między komponentami są osiągane wprost, przez wywołanie metody.

```
@Configuration
public class AppConfig {

  @Bean
  public Bean1 bean1() {
      return new Bean1();
  }

  @Bean @Qualifier("public")
  public Bean2 bean2() {
      return new Bean2(bean1());
  }
}
```

Zasięg komponentów @Scope

```
@Configuration
public class AppConfig {

  @Bean
  @Scope("prototype")
  public Bean1 bean1() {
          return new Bean1();
  }
}
```

```
@Component("helloBean")
@Scope("prototype")
public class Hello {

      // ...
}
```

Metody init i destroy

```
public class Bean {
        public void init() {
                // ...
        }
        public void destroy() {
                // ...
        }
}


@Configuration
public class AppConfig {

        @Bean(init-method = ”init”, destroy-method = ”destroy”)
        public Bean bean() {
                return new Bean();
        }
}
```

@Primary

- Jeśli wśród wielu pasujących kandydatów do @Autowire, jeden z nich jest oznaczony przez @Primary, to ten komponent będzie wybrany

@DependsOn, @Lazy

- Adnotacje te są odpowiednikami atrybutów depends-on i lazy w XML

JAVATECH

@Import i @ImportResource

- @Import – dołącza inną klasę konfiguracyjną (oznaczoną @Configuration)
- @ImportResource – dołącza standardową konfigurację w XML

Nadawanie wartości własnościom - @Value

```
@Configuration
public class AppConfig {
	private @Value("${jdbc.url}") String url;
	private @Value("${jdbc.username}") String username;
	private @Value("${jdbc.password}") String password;

	public @Bean DataSource dataSource() {
		return new DriverManagerDataSource(
			url, username, password);
	}
}
```

JAVATECH

Spring Expression Language

JAVATECH

Spring Expression Language – wprowadzenie

- Język pozwalający operować na grafie obiektów
- Interpretowany
- Używany powszechnie przez Spring
- Przyjazny XML
- Może być wykorzystany jako niezależny komponent
- Podobny do Unified EL, ale dostarczający własne rozszerzenia

JAVATECH

SpEL – przegląd możliwości

- Wyrażenia dosłowne (literalne)
- Operatory logiczne
- Wyrażenia regularne
- Porównywanie klas
- Dostęp do własności, list, tablic, słowników
- Dostęp do fasolek
- Wywołania metod i konstruktorów
- Operatory relacyjne
- Przypisania
- Operator ? :
- Zmienne

SpEL – przegląd możliwości

- Funkcje użytkownika
- Listy inline
- Filtrowanie i projekcja kolekcji
- Wyrażenia szablonowe
- ...

Hello World

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'");
String message = (String) exp.getValue();
```

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'.bytes");
byte[] bytes = (byte[]) exp.getValue();
```

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'.bytes.length");
Integer length = (Integer) exp.getValue();
```

Konwersja wyniku

- T getValue(Class<T> clazz) – dokonuje konwersji wyniku od razu do zadanej klasy
- Jeśli zachodzi taka konieczność, konwersja jest dokonywana za pomocą odpowiedniego konwertera
- Jeśli nie da się skonwertować, rzuca EvaluationException

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("new String('hello world').toUpperCase()");
String message = exp.getValue(String.class);
```

EvaluationContext i StandardEvaluationContext

- Umożliwia podanie obiektu głównego, do którego własności będą odwoływały się nazwy w wyrażeniach
- Umożliwia dostarczenie resolverów beanów, metod i zmiennych

```
Person person = new Person();
person.setName("Jan Kowalski");
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("name");
EvaluationContext context = new StandardEvaluationContext(person);
String name = exp.getValue(context, String.class);
```

```
Person person = new Person();
person.setName("Jan Kowalski");
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("name");
String name = exp.getValue(person, String.class);
```

Ustawianie własności

- Jeśli wyrażenie wskazuje na konkretną własność komponentu – możliwe jest ustawienie jej wartości
- Przy ustawianiu są uwzględniane konwertery i edytory własności

```
class Simple {
   public List<Boolean> booleanList = new ArrayList<Boolean>();
}

Simple simple = new Simple();
simple.booleanList.add(true);

StandardEvaluationContext simpleContext = new StandardEvaluationContext(simple);
parser.parseExpression("booleanList[0]").setValue(simpleContext, "false");

Boolean b = simple.booleanList.get(0);  // zwróci false
```

Użycie SpEL w konfiguracji XML

- Wyrażenia wprowadzane za pomocą #{ }
- Dostęp do beanów w projekcie
- Dostęp do własności systemowych: zmienna systemProperties

```
<bean id="randomNumber" class="lab.spring.RandomNumber">
  <property name="value" value="#{ T(java.lang.Math).random() * 100.0 }"/>
</bean>

<bean id="randomCircle" class="lab.spring.Circle">
  <property name="radius" value="#{ randomNumber.value }"/>
</bean>

<bean id="taxCalculator" class="org.spring.samples.TaxCalculator">
  <property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>
</bean>
```

Wyrażenia dosłowne

- Liczby: jak w Javie
- Wartości logiczne: true, false
- Null: null
- Łańcuchy znaków: w apostrofach: 'Hello'

Dostęp do własności, list, tablic, słowników

- Dostęp do własności poprzez nazwę
- Zapis z kropką umożliwia dostęp do złożonych własności, np: bean1.bean2.name
- W nawiasach kwadratowych indeks elementu dla tablic i list: company.employees[2]
- W nawiasach kwadratowych również dostęp do słownika: company.salary['Kowalski']

JAVATECH

Tworzenie list i tablic

- Listy:
  - {} - pusta lista
  - {1,2,3,4}
  - {{'a','b'},{'x','y'}}
- Tablice:
  - new int[4]
  - new int[]{1,2,3}
  - new int[4][5] – tablica dwuwymiarowa

JAVATECH

Metody

- Metody wykorzystują standardową składnię Javy:
  - 'abc'.substring(2, 3)
  - userManager.getCurrentUser()
- Zamiast getterów i setterów, można odwoływać się do nazwy pola bezpośrednio (mimo że prywatne):
  - userManager.currentUser

JAVATECH

Operatory

- <, >, <=, >=, ==, !=
- lt, gt, le, ge, eq, ne
- instanceof
- matches – do wyrażeń regularnych np.
  '5.00' matches '^-?\\d+(\\.\\d{2})?$'
- and, or, not (!)
- + - * /, div, %, mod
- ^ - potęgowanie
- = - przypisanie (działa tak samo jak wywołanie setValue)

Typy

- Specjalny operator T(NazwaTypu) służy do odwołania się do:
  - metod statycznych danego typu
  - samego typu jako obiekt klasy Class<NazwaTypu>

```
Class dateClass = parser.parseExpression("T(java.util.Date)").getValue(Class.class);

Class stringClass = parser.parseExpression("T(String)").getValue(Class.class);

boolean trueValue = parser.parseExpression(
    "T(java.math.RoundingMode).CEILING < T(java.math.RoundingMode).FLOOR")
    .getValue(Boolean.class);
```

JAVATECH

Konstruktory

- Konstruktory wywołuje się tak samo jak w Javie, ale wymagane jest podanie kwalifikowanej nazwy klasy:
  - new lab.spring.Person('Kowalski')
- Klasa String i typy opakowujące nie wymagają podania kwalifikowanej nazwy klasy.

JAVATECH

Selekcja

- Operator kolekcja.?[warunek] pozwala filtrować kolekcje takie jak listy i słowniki
- osoby.?[wiek <= 18] pozostawi na liście tylko te osoby, których wiek nie przekracza 18
- slownik.?[key.startsWith('a')] pozostawi w słowniku tylko te pary, których klucze zaczynają się na 'a'
- slownik.?[value.startsWith('a')] pozostawi w słowniku tylko te pary, których wartości zaczynają się na 'a'

Projekcja

- Przekształca każdy element kolekcji za pomocą zadanej funkcji
- Działa dla list
- Operatorem projekcji jest kolekcja.![przekształcenie]
- Przykład: members.![name.toLowerCase()]

Zmienne

- Definiowanie zmiennych: poprzez StandardEvaluationContext
- Dostęp do zmiennych: #nazwa

```
Person person = new Person("J. Kowalski");
StandardEvaluationContext context = new StandardEvaluationContext(person);
context.setVariable("newName", "A. Nowak");

parser.parseExpression("name = #newName").getValue(context);
System.out.println(person.getName());  // "A. Nowak"
```

#this i #root

- #this oznacza obiekt aktualnie przetwarzany (przydatne przy filtrowaniu i projekcji)
- #root oznacza zawsze obiekt główny ustawiony w EvaluationContext
- Przykład: #primes.?[#this>10]

Funkcje

- Funkcje rejestruje się w StandardEvaluationContext
- Służy do tego:
  void registerFunction(String name, Method m)
- Wywołanie funkcji – poprzedzenie jej
  nazwy znakiem # np. #mojaFunkcja(parametr1, parametr2)

JAVATECH

BeanResolver

- StandardEvaluationContext umożliwia zarejestrowanie obiektu BeanResolver:
  void setBeanResolver(BeanResolver r)
- BeanResolver jest odpowiedzialny za dostarczanie beanów
- W przeciwieństwie do zmiennych wartości są obliczane podczas obliczania wyrażenia
- Odwołanie do beana: @nazwa

JAVATECH

Inne operatory

- Operator "jeżeli to": ?:
  - false ? 'trueExp' : 'falseExp'
- Operator 'Elvis':
  - object ?: 'falseExp'
  - wtedy jeśli object != null, to zwracane jest object
- Operator bezpiecznej nawigacji: ?
  - zwraca null zamiast rzucania NPE
  - placeOfBirth?.city

Wyrażenia szablonowe

```
String randomPhrase =
    parser.parseExpression("random number is #{T(java.lang.Math).random()}",
                           new TemplateParserContext()).getValue(String.class);
```

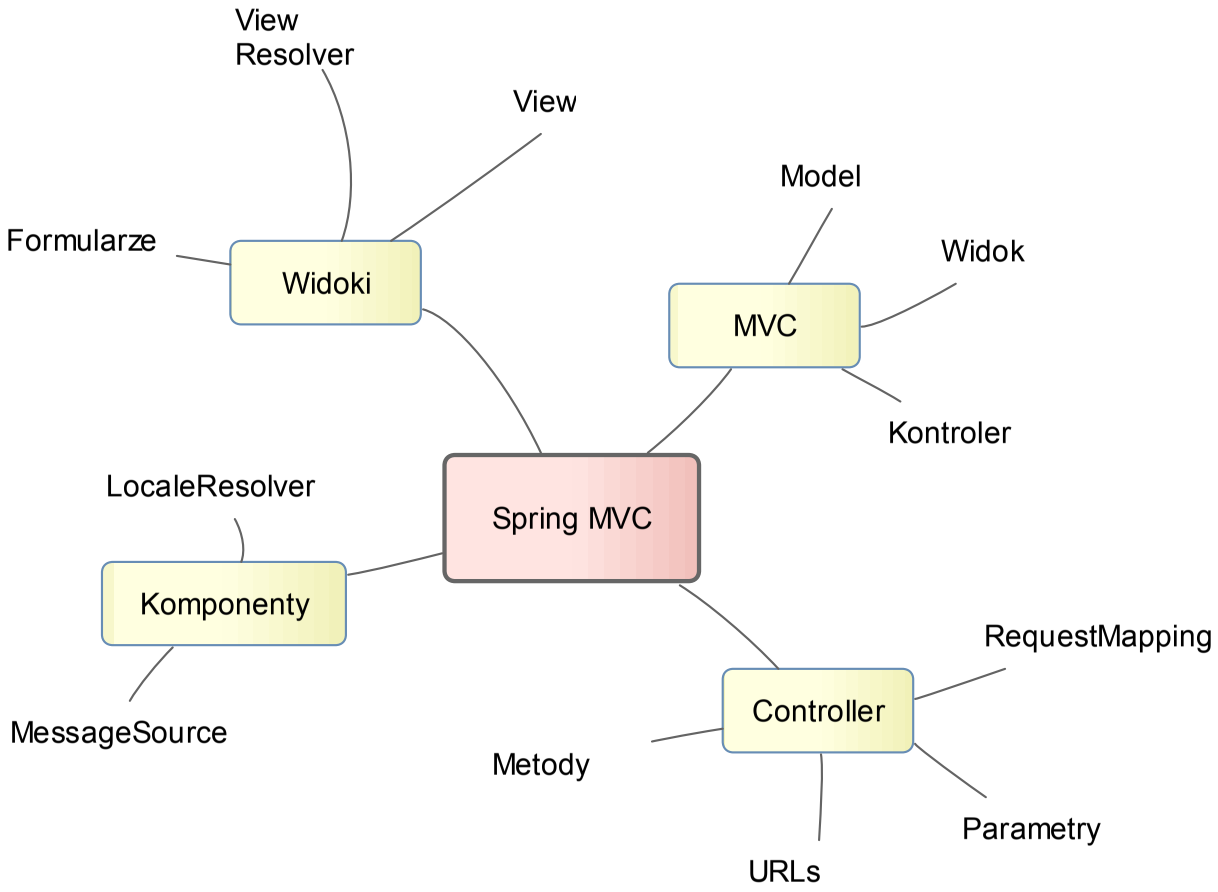
```
public class TemplateParserContext implements ParserContext {

  public String getExpressionPrefix() {
    return "#{";
  }

  public String getExpressionSuffix() {
    return "}";
  }

  public boolean isTemplate() {
    return true;
  }
}
```

JAVATECH

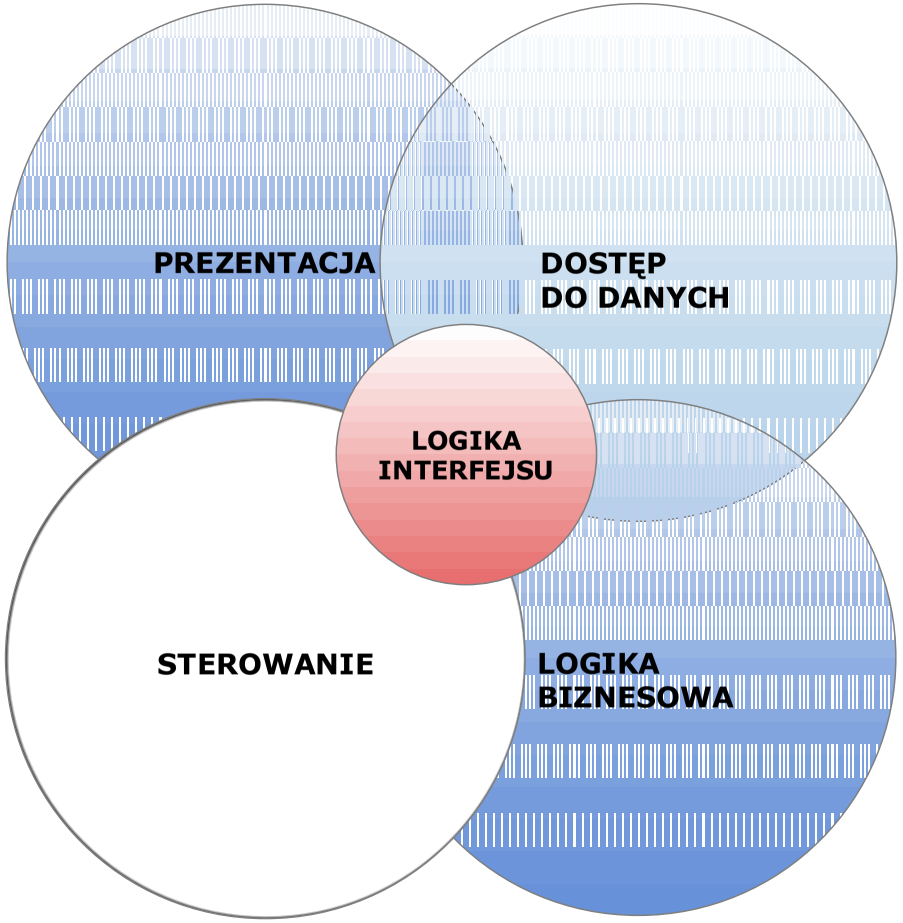
Spring MVC

Tworzenie aplikacji web

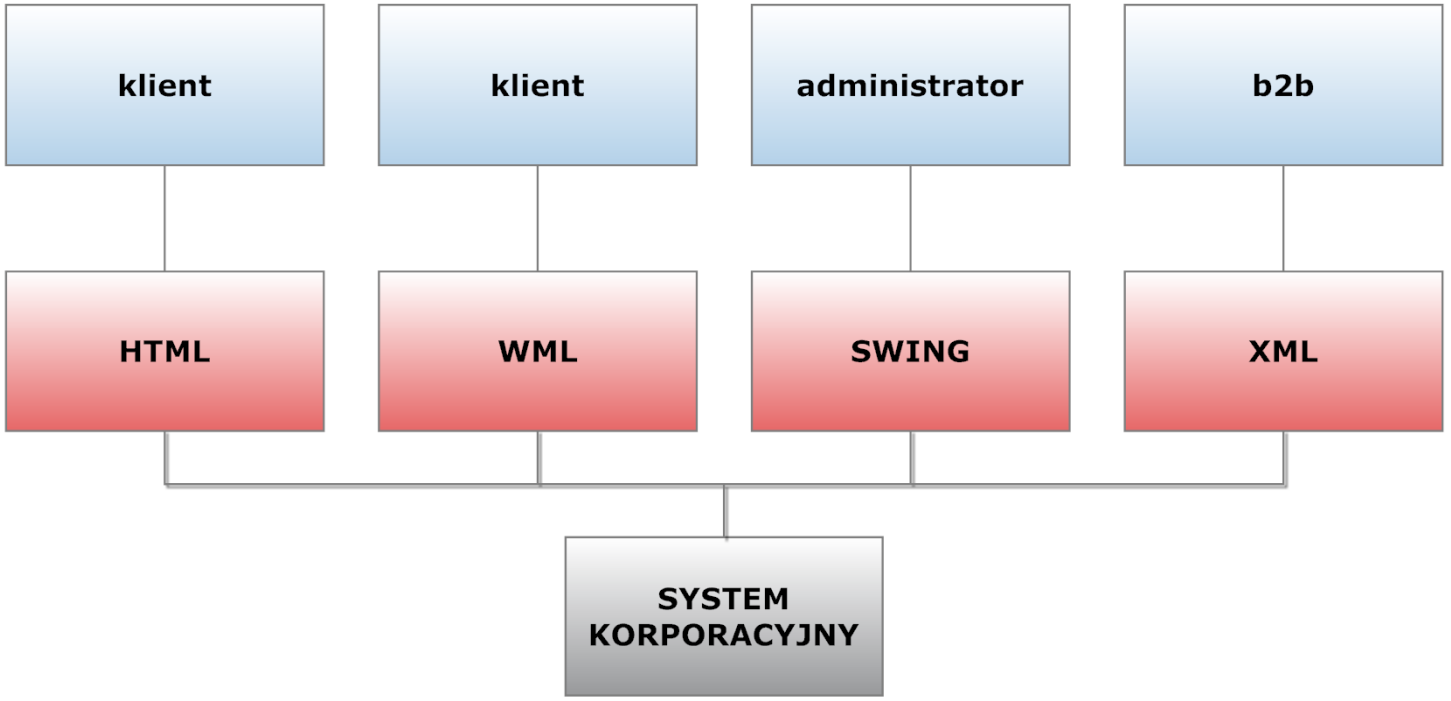
Agenda


View Resolver
View
Formularze
Widoki
Model
Widok
MVC
Kontroler
LocaleResolver
Spring MVC
Komponenty
MessageSource
RequestMapping
Controller
Metody
Parametry
URLs

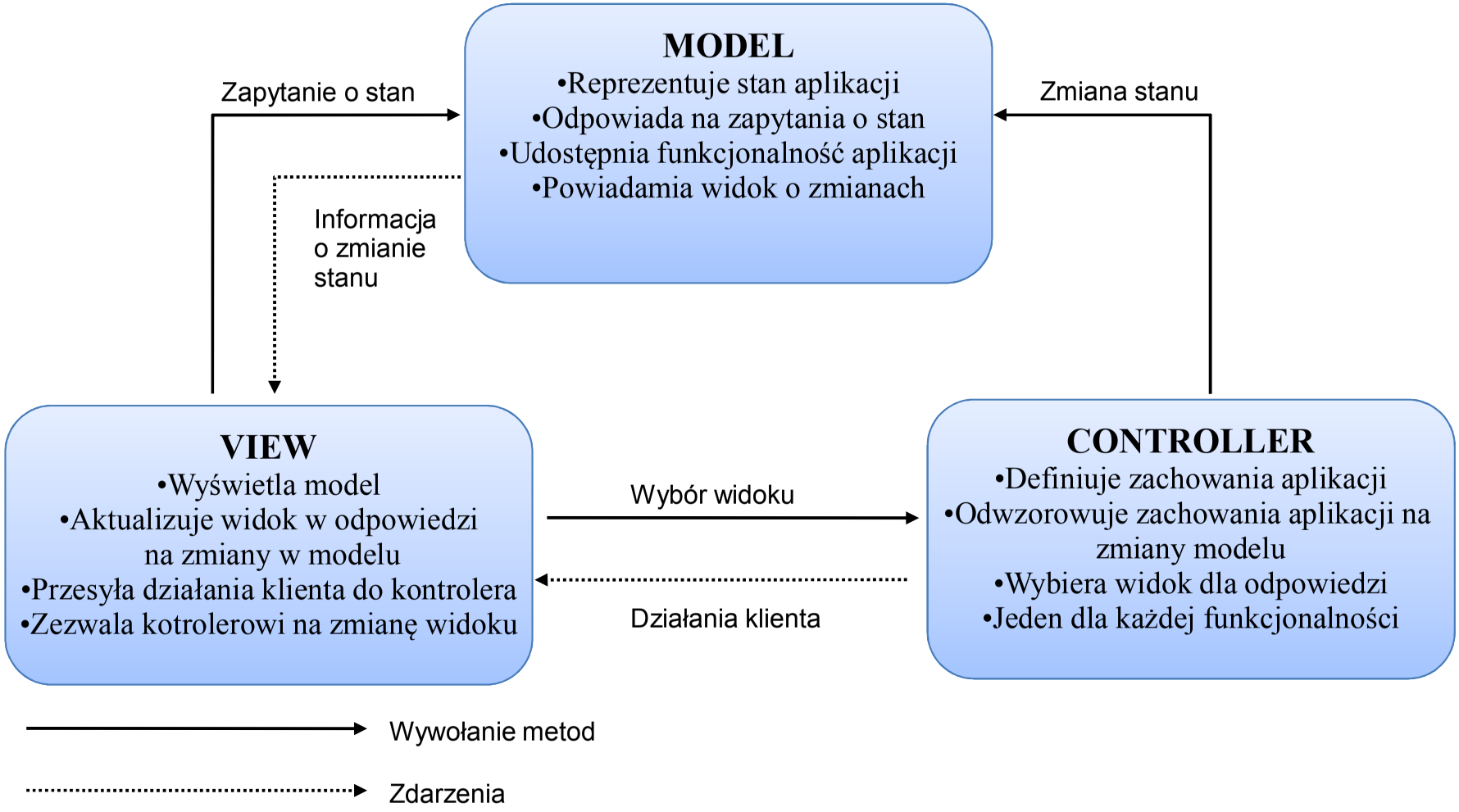
Podejście "tradycyjne"



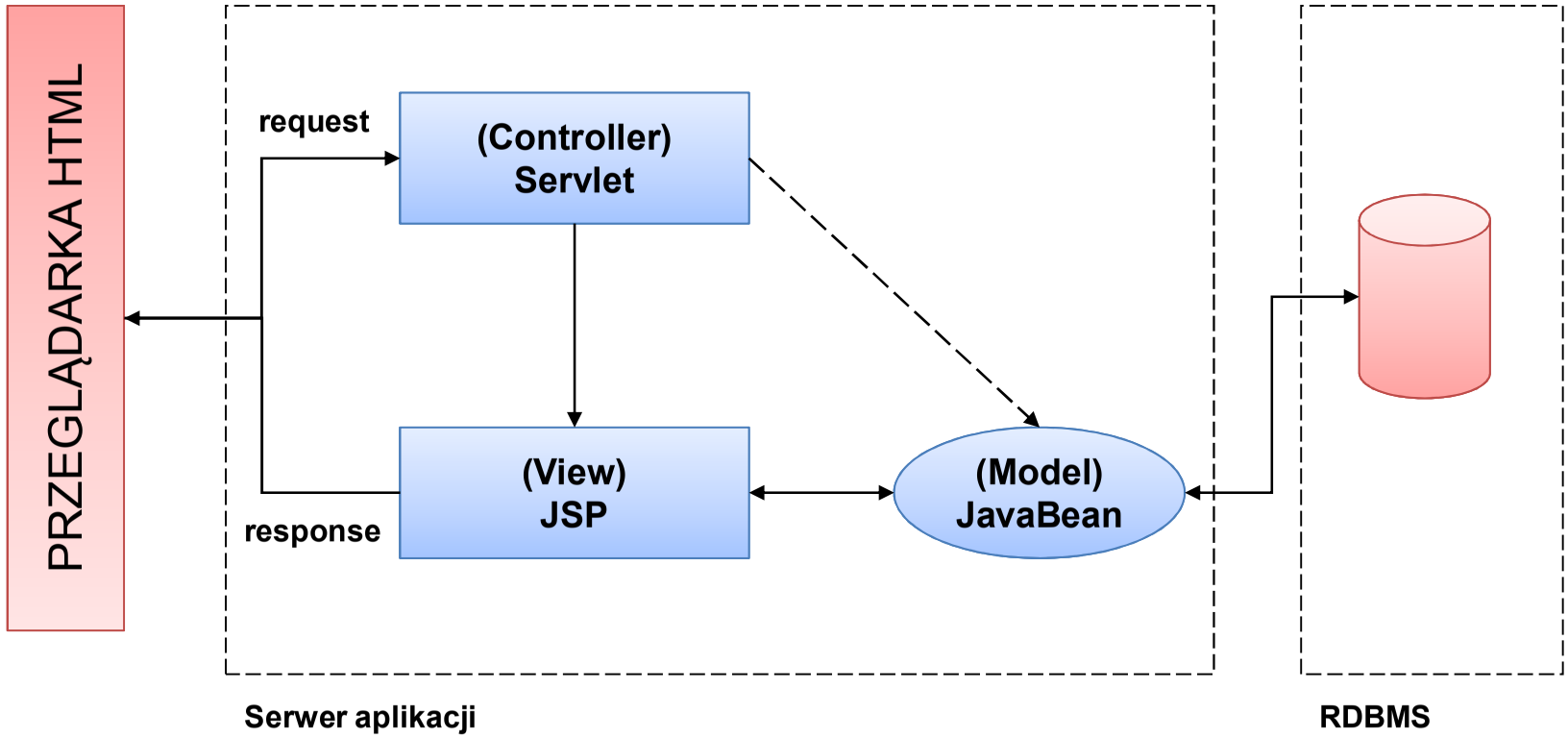
Problem do rozwiązania



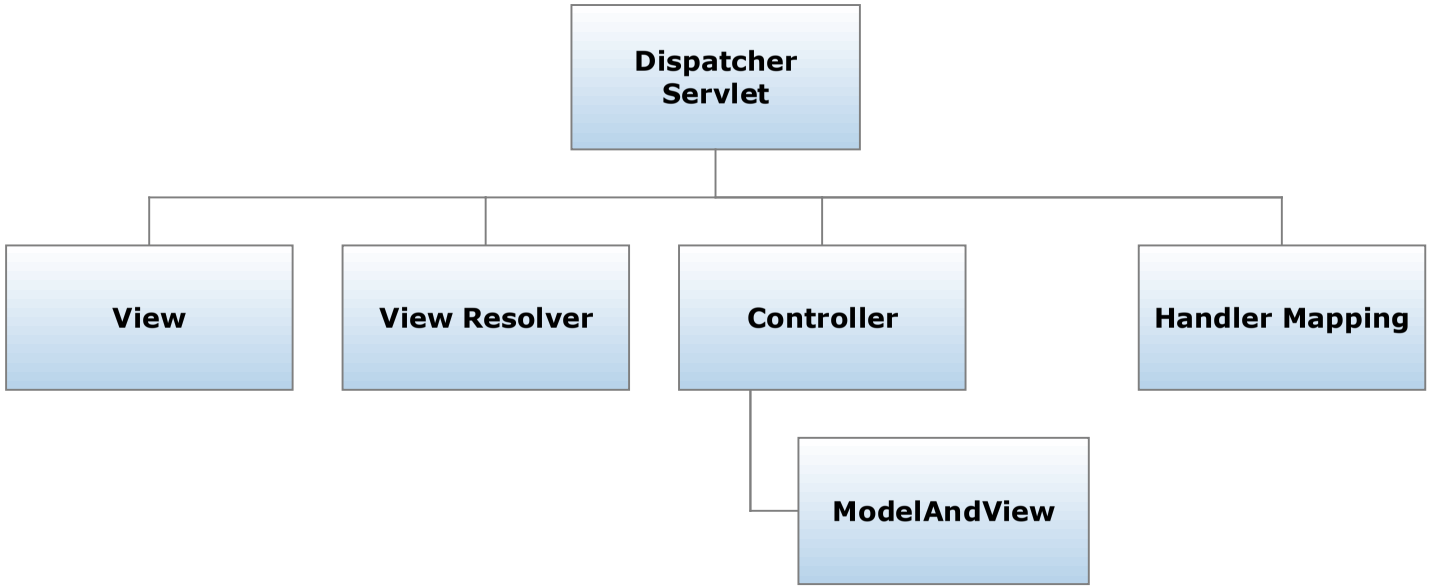
Model View Controller



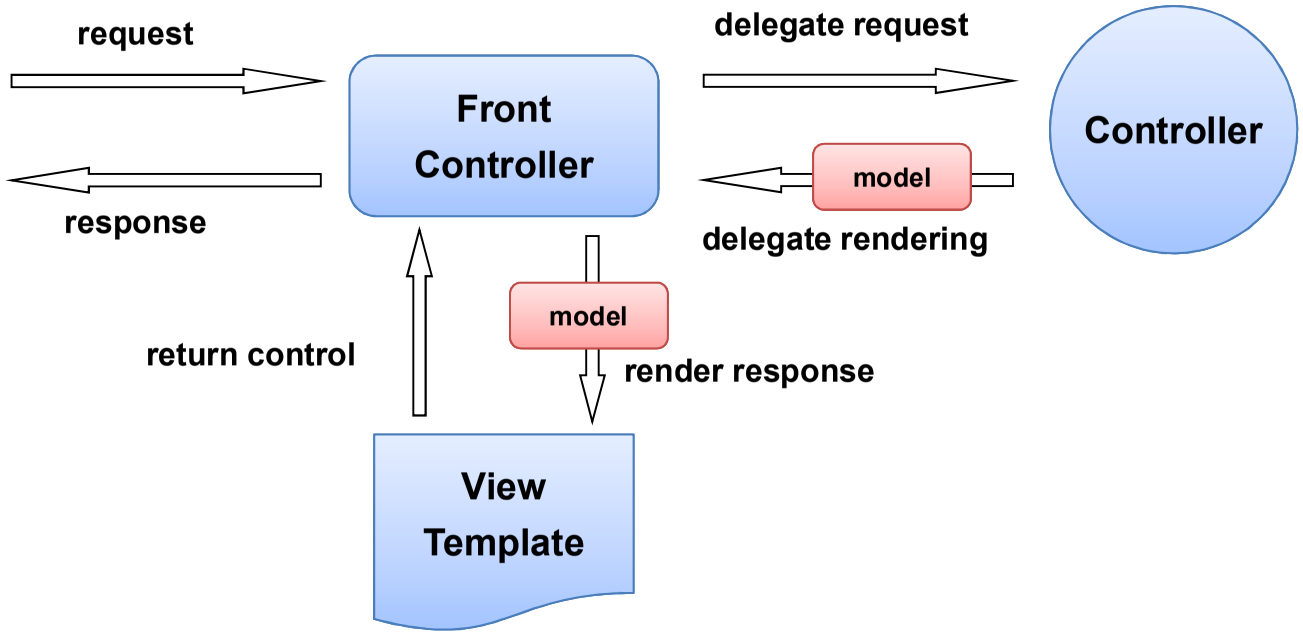
MVC w środowisku Java EE



Podstawowe komponenty Spring MVC



Przepływ danych w Spring MVC



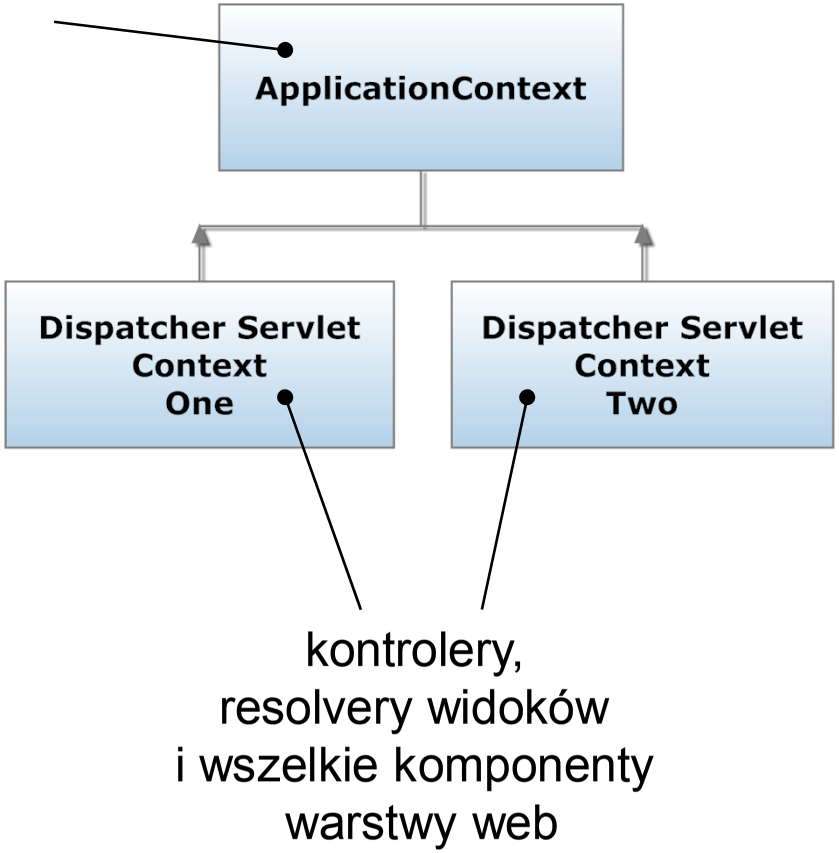
DispatcherServlet odnajduje kontroler przy pomocy obiektu HandlerMapping
DispatcherServlet przekazuje żądanie do kontrolera
Kontroler zwraca ModelAndView
DispatcherServlet odnajduje warstwę prezentacji (View) przy pomocy obiektu ViewResolver
Obiekt View prezentuje odpowiedź serwera

Konfiguracja Dispatcher Servlet

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

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

Konfiguracja kontekstu aplikacji

komponenty warstwy logiki biznesowej, warstwy danych



Konfiguracja kontekstu aplikacji

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
                /WEB-INF/conf/applicationContext.xml
                classpath:security.xml
        </param-value>
</context-param>
<listener>
        <listener-class>
                org.springframework.web.context.ContextLoaderListener
        </listener-class>
</listener>
```

Kontekst servletu

```
<servlet>
<servlet-name>dispatcher</servlet-name>
<servlet-class>
        org.springframework.web.servlet.DispatcherServlet
</servlet-class>
```

Konfiguracja kontekstu Spring skojarzonego z servletem domyślnie umieszczona jest w katalogu /WEB-INF/ w pliku o nazwie *[**servlet-name**]-servlet.xml*

/WEB-INF/***dispatcher***-servlet.xml

JAVATECH

Kontrolery w Spring 3.0

- W pierwszych wersjach Spring MVC instniała obiektowa hierarchia kontorlerów. Spring 3.0 uznaje ją za deprecated.
- W zamian wprowadza mechanizm definiowania właściwości kontrolera za pomocą adnotacji.

@Controller

- Kontrolery oznacza się adnotacją @Controller
- Dostarczają metod obsługujących żądania HTTP

```
@Controller
public class HelloWorldController {
        @RequestMapping("/helloWorld")
        public ModelAndView helloWorld() {
                ModelAndView mav = new ModelAndView();
                mav.setViewName("helloWorld");
                mav.addObject("message", "Hello World!");
                return mav;
        }
}
```

Rejestracja komponentów

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

        <context:component-scan base-package="lab.spring"/>

</beans>
```

Dzięki takiemu rozwiązaniu Spring automatycznie odnajdzie wszystkie klasy znaczone odpowiednimi adnotacjami.

@RequestMapping

- Wiąże ścieżkę w URL z danym kontrolerem.
- Może być użyte dla całej klasy jak i metody.

```
@Controller
public class HelloWorldController {
        @RequestMapping("/helloWorld")
        public ModelAndView helloWorld() {
                ModelAndView mav = new ModelAndView();
                mav.setViewName("helloWorld");

                mav.addObject("message", "Hello World!");
                return mav;
        }
}
```

JAVATECH

@RequestMapping

```
@Controller
public class HelloWorldController {
        @RequestMapping("/helloWorld")
        public ModelAndView helloWorld() {
                ModelAndView mav = new ModelAndView();
                mav.setViewName("helloWorld");

                mav.addObject("message", "Hello World!");
                return mav;
        }
}
```

@RequestMapping - parametry

- value – określa ścieżkę w URL
- method – określa metodę GET/POST/…
- params – parametry żądania HTTP (String[])
- headers – nagłówki żądania HTTP (String[])

Wszystkie elementy musza pasować jednocześnie aby metoda obsłużyła dane żądanie.

JAVATECH

Parametry metod kontrolera

- Obiekty HttpServletRequest i HttpServletResponse
- HttpSession
- org.springframework.web.context.request.WebRequest
- java.util.Locale
- java.io.InputStream / java.io.Reader
- java.io.OutputStream / java.io.Writer
- @PathVariable, @RequestParam, @RequestHeader, @RequestBody
- java.util.Map
- Command Object

JAVATECH

Wymuszenie parametrów i nagłówków

- @RequestMapping(
  value = "/form",
  method = RequestMethod.POST,
  headers="content-type=text/*")
- @RequestMapping(
  value = "/form",
  params="myParam=myValue")
- @RequestMapping(
  value = "/form",
  params={"myParam1=myValue1",
  "myParam2=myValue2"})

Parametry i nagłówki - adnotacje

- @RequestParam
- @RequestHeader

```
@RequestMapping(value = "/userInfo", method = RequestMethod.GET)
public String userInfo(@RequestParam("userId") int userId,
                       ModelMap model) {
   User user = this.user.loadUser(userId);
   model.addAttribute("user", user);
   return "userInfo";
}
```

```
@RequestMapping("/displayHeaderInfo")
        public void displayHeaderInfo(
        @RequestHeader("Accept-Encoding") String encoding,
        @RequestHeader("Keep-Alive") long keepAlive)  {
}
```

JAVATECH

Szablony URI

- Istnieje możliwość odczytu parametru z URI (nice links)
- Element URI, który chcemy traktować jako parametr konstruuje się poprzez objęcie go w nawias klamrowy {nazwa}
- Pozyskanie wartości parametru odbywa się poprzez zdefiniowanie argumentu metody kontrolera i oznaczenie go adnotacją @PathVariable("nazwa").
- Można odczytać więcej niż jeden parametr.
- Wartości parametrów są automatycznie konwertowane do odpowiedniego typu.

Odczyt parametrów

```
@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable("ownerId") String ownerId, Model model) {
  Owner owner = ownerService.findOwner(ownerId);
  model.addAttribute("owner", owner);
  return "displayOwner";
}
```

```
@RequestMapping(value="/category/{categoryId}/item/{itemId}",
                method=RequestMethod.GET)
public String getItem(@PathVariable Integer categoryId,
                      @PathVariable Integer itemId, Model model) {
  // ...
}
```

JAVATECH

Odbieranie i wysyłanie danych formularzy

- @ModelAttribute na argumencie metody kontrolera – pozwala odebrać dane formularza w postaci obiektu (bean)
- @ModelAttribute na metodzie kontrolera, pozwala umieścić obiekt zwracany przez metodę w modelu
- @ModeAttribute wymaga podania nazwy, pod którą zostanie zapisany obiekt w modelu
- Na argumencie można dodatkowo umieścić adnotację @Valid, która wymusi jego walidację
- Za argumentem trzymającym dane formularza, można umieścić argument typu BindingResult, który będzie zawierał wyniki walidacji

JAVATECH

Metody kontrolera – zwracanie wartości

- ModelAndView – obiekt zawierający w sobie zarówno model, jak i nazwę widoku, który ma zostać wyświetlony
- Model – tylko model, widoku zostanie użyty domyślny na podstawie obiektu klasy RequestToViewNameTranslator
- Map – tak jak Model, tylko podany w postaci słownika
- View – tylko widok; model zostanie utworzony na podstawie obiektów poleceń (command) oraz obiektów z adnotacją @ModelAttribute. Dodatkowo model można wzbogacić ręcznie, w ciele metody kontrolera (argument typu Model)
- String – oznacza nazwę widoku do wyświetlenia, reszta tak jak dla View
- void – jeśli metoda obsługuje odpowiedź samodzielnie, bezpośrednio pisząc do strumienia wyjściowego

JAVATECH

Metody kontrolera – zwracanie wartości

- @ResponseBody – wynik zostanie zapisany do ciała odpowiedzi HTTP, po ewentualnej konwersji
- HttpEntity<?> lub ResponseEntity<?>
- Obiekt dowolnego innego typu – wtedy metoda musi być adnotowana @ModelAttribute – obiekt ten zostanie udostępniony w modelu pod wskazaną nazwą

Kontrola wiązania danych: @InitBinder

- Metoda kontrolera oznaczona @InitBinder otrzymuje jako argument obiekt typu WebDataBinder
- WebDataBinder pozwala:
  - zarejestrować dodatkowe edytory własności do konwersji danych
  - ustawić walidator (implementujący Validator, nie JSR-303)

```
@Controller
public class MyFormController {
   @InitBinder
   public void initBinder(WebDataBinder binder) {
      SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
      dateFormat.setLenient(false);
      binder.registerCustomEditor(Date.class,
         new CustomDateEditor(dateFormat, false));
   }
}
```

JAVATECH

Przechwytywanie żądań (intercepting)

- Interfejs HandlerInterceptor:
  - preHandle – zanim zostanie wywołana metoda kontrolera; jeśli zwróci false, żądanie nie jest dalej przetwarzane
  - postHandle – po wywołaniu metody kontrolera, ale zanim zostanie zarenderowany widok
  - afterCompletion – po zarenderowaniu widoku
- HandlerInterceptorAdapter dla wygody umożliwia implementację tylko wybranych metod
- Jedno żądanie może przechodzić przez wiele interceptorów

Przechwytywanie żądań – interceptor

```
public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

    private int openingTime;
    private int closingTime;

    public void setOpeningTime(int openingTime) {
        this.openingTime = openingTime;
    }

    public void setClosingTime(int closingTime) {
        this.closingTime = closingTime;
    }

    public boolean preHandle(
            HttpServletRequest request,
            HttpServletResponse response,
            Object handler) throws Exception {

        Calendar cal = Calendar.getInstance();
        int hour = cal.get(Calendar.HOUR_OF_DAY);
        if (openingTime <= hour && hour < closingTime) {
            return true;
        } else {
            response.sendRedirect("http://host.com/outsideOfficeHours.html");
            return false;
        }
    }
}
```

Rejestracja interceptorów

```
<mvc:interceptors>
  <bean id="officeHoursInterceptor"
        class="lab.spring.TimeBasedAccessInterceptor">
    <property name="openingTime" value="9"/>
    <property name="closingTime" value="18"/>
  </bean>

  <bean ... />
  <bean ... />
</mvc:interceptors>
```

ViewResolver i View

- ViewResolver zwraca widok (obiekt klasy View) na podstawie nazwy widoku i lokalizacji
- ViewResolver może zwrócić null
- View renderuje model
- Mechanizm ten umożliwia używanie różnych technologii renderowania: JSP, JSTL, Velocity, FreeMarker etc.

```
public interface ViewResolver {
        View resolveViewName(String viewName, Locale locale);
}

public interface View {
        String    getContentType();

        void render(
                Map<String,?> model,
                HttpServletRequest request,
                HttpServletResponse response);
}
```

Standardowe implementacje ViewResolver

- AbstractCachingResolver
  - buforuje widoki w celu zmniejszenia nakładu na ich przygotowanie
- XmlViewResolver
  - konfiguruje widoki z pliku XML z definicjami beanów
  - poszczególne beany reprezentują widoki
  - domyślnie wczytuje definicje z pliku: /WEB-INF/views.xml
- ResourceBundleViewResolver
  - konfiguruje widoki zapisane w pliku zasobów jako:
  - do

```
[nazwa-widoku].(class) = nazwa-klasy-widoku
[nazwa-widoku].url = url-widoku
```

Standardowe implementacje ViewResolver

- UrlBasedViewResolver – w prosty sposób zamienia nazwy widoków na widoki dla opowiadających URLi
- InternalResourceViewResolver –
  - podklasa UrlBasedViewResolver
  - obsługuje JstlView i TilesView
- VelocityViewResolver / FreeMarkerViewResolver -
  - podklasy UrlBasedViewResolver
  - obsługują VelocityView i FreeMarkerView
- ContentNegotiatingViewResolver –
  - wybiera inny resolwer widoków na podstawie żądanego pliku oraz nagłówka Accept

Konfiguracja wielu resolwerów widoków

- W kontekście aplikacji można skonfigurować wiele resolwerów
- Własność "order" określa, w jakiej kolejności mają być uwzględniane
- Jeśli nie zostanie znaleziony żaden pasujący resolwer, Spring rzuci ServletException

JAVATECH

Przekierowania

- Kontroler może zwrócić instancję RedirectView
  - RedirectView wywoła HttpServletResponse.sendRedirect()
- redirect:URL
  - przekierowuje na dany URL
  - kontroler nie musi w ogóle wiedzieć, że następuje przekierowanie – może dostać taką nazwę z zewnątrz
- forward:URL
  - dokonuje wewnętrznego przekierowania, które nie skutkuje dodatkowym żądaniem HTTP

JAVATECH

ContentNegotiatingViewResolver

- Deleguje generowanie widoku do innych resolwerów
- Porównuje zawartość nagłówka Accept z Content-Type widoków kolejno zwracanych przez resolwery
- Jeśli żaden widok nie pasuje, wybiera pasujący widok z listy domyślnych widoków
- Parametry:
  - mediaTypes – mapowanie rozszerzeń plików na content-type
  - viewResolvers – lista resolwerów widoków
  - defaultViews – lista domyślnych widoków

ContentNegotiatingViewResolver – przykład

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
  <property name="mediaTypes">
    <map>
      <entry key="atom" value="application/atom+xml"/>
      <entry key="html" value="text/html"/>
      <entry key="json" value="application/json"/>
    </map>
  </property>
  <property name="viewResolvers">
    <list>
      <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
      <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
      </bean>
    </list>
  </property>
  <property name="defaultViews">
    <list>
      <bean class="org.springframework.web.servlet.view.json.MappingJacksonJsonView" />
    </list>
  </property>
</bean>

<bean id="content" class="com.springsource.samples.rest.SampleContentAtomView"/>
```

LocaleResolver

- Ustawia właściwe Locale na podstawie parametrów żądania
- Locale można pobrać z dowolnego miejsca za pomocą RequestContext.getLocale
- Locale może być też dostarczone jako argument metody kontrolera
- Dodatkowo umożliwia ręczne ustawienie Locale, poprzez metodę setLocale

JAVATECH

Dostępne implementacje LocaleResolver

- AcceptHeaderLocaleResolver – ustawia Locale na podstawie nagłówka "accept-language"
- CookieLocaleResolver – ustawia Locale na podstawie ciasteczka przechowywanego po stronie klienta
- SessionLocaleResolver – przechowuje ustawienia Locale w sesji HTTP
- FixedLocaleResolver – zwraca zawsze domyślne Locale ustawione przez JVM; Locale nie może być zmienione

CookieLocaleResolver – przykład

```
<bean id="localeResolver"
        class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

    <property name="cookieName" value="clientlanguage"/>

    <!-- w sekundach -->
    <property name="cookieMaxAge" value="100000">

</bean>
```

LocaleChangeInterceptor

- Umożliwia wygodne przełączanie Locale za pomocą parametru żądania HTTP
- Poniższy kod umożliwia przełączanie Locale za pomocą dodania "lang=PL_pl" do URL:

```
<mvc:interceptors>
      <list>
         <ref bean="localeChangeInterceptor"/>
      </list>
</mvc:interceptors>

<bean id="localeChangeInterceptor"
     class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
   <property name="paramName" value="lang"/>
</bean>

<bean id="localeResolver"
     class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>
```

JAVATECH

Tematy – Themes

- Temat określa zestaw statycznych zasobów kontrolujących wygląd strony:
  - grafika
  - CSS
  - komunikaty tekstowe
- Można dostarczyć wiele tematów dla jednej aplikacji i pozwolić użytkownikowi wybrać jeden
- Każdy temat jest określony osobnym zestawem plików css/grafik
- Każdy temat posiada osobny plik konfiguracyjny wskazujący odpowiednie pliki (składnia taka sama jak dla zasobów zawierających komunikaty odczytywane przez MessageSource)

Pliki konfiguracyjne tematów

- dark.properties:

```
css=themes/dark.css
page.title=Ciemna Strona Springa
welcome.message=Witaj! Dobrej nocy!
```

- bright.properties:

```
css=themes/bright.css
page.title=Jasna Strona Springa
welcome.message=Dzień Dobry!
```

ThemeSource i ThemeResolver

- ThemeSource określa skąd wziąć opisy tematów – w tym przypadku ResourceBundleThemeSource wczytuje je z plików .properties
- ThemeResolver określa, którego tematu użyć. Dostępne są standardowo:
  - FixedThemeResolver,
  - SessionThemeResolver,
  - CookieThemeResolver

```
<bean id="themeSource"
    class="org.springframework.ui.context.support.ResourceBundleThemeSource"/>
<bean id="themeResolver"
    class="org.springframework.web.servlet.theme.FixedThemeResolver">
        <property name="defaultThemeName" value="bright"/>
</bean>
```

Własny ThemeResolver

```
public class DarkAndBrightThemeResolver extends AbstractThemeResolver {

    @Override
    public String resolveThemeName(HttpServletRequest arg0) {
        return isNight() ? "dark" : "bright";
    }

    private boolean isNight() {
      Calendar cal = Calendar.getInstance();
      int hour = cal.get(Calendar.HOUR_OF_DAY);
      return hour < 6 || hour > 22;
    }

    @Override
    public void setThemeName(HttpServletRequest arg0,
                             HttpServletResponse arg1, String arg2) {
    }
}
```

Wykorzystanie tematu w widoku

- <spring:theme code="[klucz w pliku properties tematu]" />

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>

<html>
  <head>
    <link rel="stylesheet" href='<spring:theme code="css"/>' type="text/css" />
    <title><spring:theme code="page.title"/></title>
  </head>
  <body>
    <spring:theme code="welcome.message" />
  </body>
</html>
```

ThemeChangeInterceptor

- Działa analogicznie jak LocaleChangeInterceptor
- Właśność paramName określa nazwę parametru żądania, który zawiera nazwę tematu

Upload plików

- Spring sam z siebie nie realizuje uploadu plików
- Wymagane użycie biblioteki Commons FileUpload
- Spring dostarcza komponent do wygodnej integracji: CommonsMultipartResolver
- CommonsMultipartResolver wykrywa żądania typu multipart i opakowuje HttpServletRequest w MultipartHttpRequest
- Spring udostępnia przyjęte pliki pod postacią MultipartFile

```
<bean id="multipartResolver"
   class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

   <property name="maxUploadSize" value="100000"/>
   <property name="maxInMemorySize" value="10000"/>

</bean>
```

Upload plików – formularz i kontroler

```
<html>
   <head><title>Upload a file</title></head>
   <body>
      <form method="post" action="/form" enctype="multipart/form-data">
         <input type="text" name="name"/>
         <input type="file" name="file"/>
         <input type="submit"/>
      </form>
   </body>
</html>
```

```
@Controller
public class FileUpoadController {

   @RequestMapping(value = "/form", method = RequestMethod.POST)
   public String handleFormUpload(@RequestParam("name") String name,
      @RequestParam("file") MultipartFile file) {
      try {
          byte[] bytes = file.getBytes();
          return "redirect:uploadSuccess";
                      } catch (IOException e) {
                 return "redirect:uploadFailure";
                      }
   }
```

JAVATECH

Obsługa wyjątków

- Metoda kontrolera może się nie powieść
- Domyślnie kontener zwróci HTTP 500 Internal Server Error i wypisze stacktrace na stronie
- Jak to naprawić?
  - obsłużyć błąd w metodzie kontrolera, tak aby żaden wyjątek się nie wydostał na zewnątrz – niewygodne – try/catch dla każdej metody
  - umieścić osobną metodę obsługi błędów w kontrolerze – za pomocą adnotacji @ExceptionHandler
  - skonfigurować SimpleMappingExceptionResolver
  - zaimplementować własny HandlerExceptionResolver, a w nim metodę resolveException, która zwraca odpowiednią stronę

JAVATECH

@ExceptionHandler

```
@Controller
public class SimpleController {

    @RequestMapping(value = "/form", method = RequestMethod.POST)
    public String handleFormUpload(@RequestParam("name") String name,
        @RequestParam("file") MultipartFile file) throws IOException {

        byte[] bytes = file.getBytes();
        return "redirect:uploadSuccess";
    }

    @ExceptionHandler(IOException.class)
    public String handleIOException(IOException ex, HttpServletRequest request) {
      return ClassUtils.getShortName(ex.getClass());
    }
}
```

SimpleMappingExceptionResolver

- Pozwala określić, jaki widok ma zostać wyświetlony dla danego wyjątku
- Umożliwia skonfigurowanie domyślnego widoku dla wszystkich pozostałych błędów

```
<bean
        class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
        <property name="exceptionMappings">
                <map>
                        <entry key="DataAccessException" value="data-error" />
                        <entry key="IOException" value="io-error" />
                </map>
        </property>
        <property name="defaultErrorView" value="general-error" />
</bean>
```

Własny HandlerExceptionResolver

```
public class IOExceptionResolver implements HandlerExceptionResolver {
        private int order;

        public ModelAndView resolveException(HttpServletRequest request,
                                             HttpServletResponse response,
                                             Object handler,
                                             Exception e) {
          if (e instanceof IOException)
            return new ModelAndView("ioExceptionView");
          else
            return null;
        }
}
```

Convention over configuration

- ModelMap i ModelAndView reprezentują model w postaci słownika
- Przy dodawaniu obiektów, nazwy kluczy mogą być pominięte
- Spring wydedukuje nazwy na podstawie nazwy klasy

```
public ModelAndView handleRequest(HttpServletRequest request,
                                  HttpServletResponse response) {

    List<CartItem> cartItems = ...
    User user = ...

    ModelAndView mav = new ModelAndView("displayShoppingCart");

    mav.addObject(cartItems); // zostanie dodane pod kluczem "cartItemList"
    mav.addObject(user); // zostanie dodane pod kluczem "user"

    return mav;
}
```

ModelAndView - zasady tworzenia nazw kluczy

- pojedyncze obiekty – jak nazwa klasy, tylko z małej litery
- java.util.HashMap – "hashMap"
- java.util.List, java.util.Set lub tablica – nazwa klasy pierwszego elementu na liście + "List"
- pusta lista, zbiór lub tablica nie zostanie dodana w ogóle
- próba dodania null – IllegalArgumentException

JAVATECH

RequestToViewNameTranslator

- Metoda kontrolera nie musi zwracać explicite nazwy widoku
- W tej sytuacji decyzję podejmuje instancja implementująca RequestToViewNameTranslator, np. DefaultRequestToViewNameTranslator
- Domyślnie bierze ona ścieżkę w URI i usuwa z niej wiodący "/" oraz rozszerzenie:
  - http://localhost:8080/gamecast/display.html -> display
  - http://localhost:8080/gamecast/displayShoppingCart.html -> displayShoppingCart
  - http://localhost:8080/gamecast/admin/index.html -> admin/index
- Można zaimplementować własny RequestToViewNameTranslator

Elementy konfiguracji Spring MVC

- mvc:annotation-driven
  - Uruchamia wsparcie dla konwersji i formatowania za pomocą adnotacji
  - Uruchamia walidację JSR-303
  - Umożliwia podpięcie własnego ConversionService
- mvc:view-controller
  - Definiuje kontroler, który zawsze przekierowuje do wskazanego widoku

```
<mvc:view-controller path="/" view-name="home"/>
```

Elementy konfiguracji Spring MVC

- mvc:resources
  - Pozwala serwować statyczne pliki z wybranego katalogu, za pośrednictwem DispatcherServlet

```
<mvc:view-resource mapping="/resources/**" location="/public-resources/"/>
```

- mvc:default-servlet-handler
  - Pozwala dowiązać DispatcherServlet do "/" zachowując możliwość korzystania z domyślnego serwletu kontenera

```
<mvc:default-servlet-handler/>
```

JAVATECH

Techniki renderowania widoków

Biblioteka znaczników JSP

Przegląd technik renderowania widoków

- JSP/JSTL
- Tiles
- Velocity
- FreeMarker
- XSLT
- Document views (PDF, Excel)
- Jasper
- Feed
- XML
- JSON

JSP/JSTL - Konfiguracja

- Wymagany kontener skonfigurowany do pracy z JSP/JSTL
- W przypadku użycia Jetty, wymagane jest umieszczenie bibliotek JSP i JSTL na classpath (w dystrybucji Jetty)
- Konfiguracja viewResolver:

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

- Nazwy widoków odpowiadają nazwom stron JSP (bez rozszerzenia)

Odwoływanie się do modelu i beanów

- Odwoływanie się do modelu poprzez ${nazwa-klucza}
- Możliwość wykonywania wyrażeń Spring EL (od Spring 3.0.1):
  - <spring:eval expression="..."/>

Biblioteka tagów JSP Spring

- Zawartość:
  - form
  - input
  - checkbox, checkboxes
  - radiobutton, radiobuttons
  - password
  - select
  - option, options
  - textarea
  - hidden
  - errors

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
```

Tagi form i input

- form – renderuje formularz HTML
- input – renderuje standardowy element do wprowadzania tekstu
- commandName – klucz pod jakim zostanie w modelu zapisany obiekt z danymi formularza; domyślnie "command"
- path – nazwa identyfikująca własności JavaBean obiektu z danymi formularza; może odnosić się do pola zagnieżdżonego

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags/form" %>

<spring:form commandName="person">
  <table>
    <tr><td>First Name:</td><td><spring:input path="firstName"/></td></tr>
    <tr><td>Last Name:</td><td><spring:input path="lastName"/></td></tr>
    <tr><td colspan="2"><input type="submit" value="Save Changes" /></td></tr>
  </table>
</form:form>
```

Tag checkbox

- Renderuje obiekt HTML "input" typu "checkbox"
- Własność obiektu powiązana z tym polem może być:
  - typu boolean
  - kolekcją lub tablicą – wtedy z każdą kontrolką checkbox jest związana jakaś wartość; checkbox jest zaznaczony, jeśli ta wartość występuje w kolekcji

```
<spring:checkbox path="user.languages" value="English"/>
<spring:checkbox path="user.languages" value="Polish"/>
```

  - dowolnym innym obiektem – checkbox jest zaznaczony, jeśli związana z nim wartość jest równa wartości własności

```
<spring:checkbox path="user.langauge" value="English"/>
```

Tag checkboxes

- Generuje listę kontrolek typu checkbox
- Dane do listy są pobierane z modelu, z klucza identyfikowanego atrybutem "items"
- Dane mogą być podane w postaci słownika (Map)
  - klucz oznacza wartość związaną z kontrolką
  - wartość oznacza etykietę kontrolki
- Dane mogą być podane też w postaci listy. Wtedy pola "itemLabel" i "itemValue" określają, z których własności elementu listy pobierana jest etykieta i wartość.

```
<spring:checkboxes path="user.languages"
                   items="${availableLanguages}"
                   itemValue="languageCode"
                   itemLabel="languageDescription" />
```

Tagi radiobutton i radiobuttons

- Umożliwiają wybranie tylko jednej pozycji
- Przekazują wybraną wartość do/z modelu
- Tag radiobuttons działa analogicznie jak checkboxes

```
<spring:radiobutton path="sex" value="M"/>
<spring:radiobutton path="sex" value="F"/>
```

```
<spring:radiobuttons path="sex" items="${sexOptions}"/>
```

Tag password

- Tak samo jak input, ale maskuje wpisywane znaki
- Domyślnie hasło nie jest wczytywane z modelu
- Jeśli hasło ma być wczytywane, należy ustawić showPassword = "true"

Tag select i option

- Renderuje "drop-down box", tj. rozwijalną listę wyboru
- Dane do listy są ładowane z modelu lub z zagnieżdżonych znaczników option

```
<spring:select path="skills" items="${skills}"/>
```

```
<spring:select path="skills">
  <spring:option value="C++"/>
  <spring:option value="Java"/>
  <spring:option value="Scala"/>
</spring:select>
```

Tagi textarea i hidden

- Textarea:

```
<spring:textarea path="notes" rows="3" cols="20" />
```

- Pole ukryte:

```
<spring:hidden path="hiddenProperty"  />
```

Tag errors – wyświetlanie błędów walidacji

- Wyświetla komunikat błędu związany z polem podanym w path
- Może wyświetlić wszystkie błędy – wtedy jako ścieżkę podajemy "*"
- Komunikaty są wyświetlane wewnątrz
- Własność element umożliwia zmianę span na np. div
- Własność cssStyle umożliwia graficzne wyróżnienie błędów

```
<form:form>
    <form:errors path="*" cssClass="errorBox" />
    <table>
      <tr>
        <td>First Name:</td>
        <td><form:input path="firstName" /></td>
        <td><form:errors path="firstName" /></td>
      </tr>
      ...
    </table>
</form:form>
```

JAVATECH

Walidacja i konwersja

Walidacja danych

- Mechanizm walidacji niezależny od warstwy
- Interfejs Validator
- Opcjonalnie walidacja za pomocą adnotacji (JSR-303)

Interfejs Validator

- public boolean supports(Class clazz)

  dostarcza informacji, które klasy są "obsługiwane"

- public void validate(Object target, Errors errors)

  dokonuje walidacji, zapisując ewentualne błędy do obiektu errors

Validator – przykład

```
public class Person {
  private String name;
  private int age;
  // ... gettery, settery itp.
}

public class PersonValidator implements Validator {

    public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }

    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "too.old");
        }
    }
}
```

JAVATECH

ValidationUtils

- invokeValidator(Validator validator, Object obj, Errors errors) – deleguje walidację obiektu do innego walidatora – przydatne do walidowania złożonych obiektów

- rejectIfEmpty(Errors errors, String field, String errorCode) – sprawdza, czy pole tekstowe jest puste

- rejectIfEmptyOrWhitespace(Errors errors, String field, String errorCode) – odrzuca pole tekstowe jeśli jest puste lub zawiera jednie znaki białe (spacje, taby itp).

JAVATECH

Zgałaszanie błędów walidacji za pomocą Errors

- Do zgłaszania błędów globalnych:
  - void reject(String errorCode)
  - void reject(String errorCode, String defaultMessage)
  - void reject(String errorCode, Object[] errorArgs, String defaultMessage)

- Do zgłaszania błędów związanych z pojedynczym polem:
  - void rejectValue(String field, String errorCode)
  - void rejectValue(String field, String errorCode, String defaultMessage)
  - void rejectValue(String field, String errorCode, Object[] errorArgs, String defaultMessage)

Walidacja za pomocą adnotacji JSR-303

```
public class Person {

  @NotNull
  @Size(max=64)
  private String name;

  @Min(0)
  @Max(110)
  private int age;

}
```

JAVATECH

Walidacja za pomocą JSR-303 – adnotacje

- NotNull
- Min, Max – wartość minimalna i maksymalna (dla liczb)
- Size(min=, max=) – ograniczenie wielkości tekstu, kolekcji
- Future, Past – sprawdza, czy data jest w przyszłości/przeszłości
- Digits – nakłada ograniczenia na liczbę cyfr
- Valid – rekursywne sprawdzenie obiektu
- EMail – niestandardowe, tylko w Hibernate Validator
- NotEmpty – niestandardowe, tylko w Hibernate Validator

JSR-303 – konfiguracja

- Umieścić na classpath bibliotekę implementującą JSR-303 np. Hibernate Validator 4
- Dołączyć do deskryptora XML:

```
<bean id="validator"
    class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

- LocalValidatorFactoryBean dostarcza implementację interfejsu Validator, delegującą walidację do JSR-303

JSR-303 – własne ograniczenia

- Zaimplementować interfejs:
  - ConstraintValidator<A extends java.lang.annotation.Annotation,T>
- Interfejs ten definiuje dwie metody:
  - void initialize(A annotation)
  - boolean isValid(T value, ConstraintValidatorContext ctx)
- Utworzyć nową adnotację:

```
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy=MyConstraintValidator.class)
public @interface MyConstraint {
}
```

BeanWrapper i BeanWrapperImpl

- Umożliwiają wygodny zapis i odczyt pól fasolek
- Dokonują odpowiednich konwersji z/na String
- Umożliwiają zapis i odczyt pól złożonych oraz kolekcji
- Umożliwiają rejestrowanie własnych konwersji
- Wykorzystywane głównie przez BeanFactory, ale dostępne dla programisty

BeanWrapper – składnia dostępu do pól

- age – odwołanie do pola age (setAge, getAge)
- account.value – odwołanie do pola zagnieżdżonego
- account[2] – trzeci element tablicy/listy lub innej naturalnie uporządkowanej kolekcji
- account[companyName] – element pod kluczem "companyName" a słowniku account

BeanWrapper – przykład użycia w kodzie

```
public class Company {
    private String name;
    private Employee managingDirector;
    // ... gettery i settery
}

public class Employee {
    private String name;
    private float salary;
            // ... gettery i settery
}
```

```
BeanWrapper company = BeanWrapperImpl(new Company());
company.setPropertyValue("name", "Some Company Inc.");

BeanWrapper jim = BeanWrapperImpl(new Employee());
jim.setPropertyValue("name", "Jim Stravinsky");
company.setPropertyValue("managingDirector", jim.getWrappedInstance());

Float salary = (Float) company.getPropertyValue("managingDirector.salary");
```

Domyślne edytory właściwości

- ByteArrayPropertyEditor
- ClassEditor
- CustomBooleanEditor
- CustomCollectionEditor
- CustomDateEditor – nie rejestrowany domyślnie
- CustomNumberEditor – Integer, Long, Float, Double
- FileEditor
- InputStreamEditor – format: [język]_[państwo]_[wariant]
- LocaleEditor
- PatternEditor
- PropertiesEditor
- StringTrimmerEditor – nie rejestrowany domyślnie
- URLEditor

Definiowanie własnych edytorów własności

- Nazwać klasę: <KlasaWłasności>Editor
- Rozszerzyć klasę PropertyEditorSupport
- Zaimplementować metody:
  - void setAsText(String str)
  - String getAsText() (opcjonalnie)
- Umieścić implementację w tym samym pakiecie co klasa własności

Definiowanie własnych edytorów własności (2)

- CustomEditorConfigurer – umożliwia zarejestrować dowolny edytor dla dowolnego typu własności

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
 <property name="customEditors">
  <map>
   <entry key="example.ExoticType" value="example.ExoticTypeEditor"/>
  </map>
 </property>
</bean>
```

Konwersje typów w Spring 3

- Converter<S, T>
  - prosty konwerter przekształcający obiekty typu S w T
- ConverterFactory<S, R> -
  - zwraca konwertery przekształcające obiekty typu S w obiekty pochodne od R;
  - umożliwia obsługę całej hierarchii klas (po stronie obiektu docelowego).
- GenericConverter – uniwersalny konwerter, może przekształcać cokolwiek w cokolwiek innego;
  - ma największe możliwości,
  - ma odstęp do deklaracji przekształcanych pól (np. do adnotacji)
  - implementacja wymaga najwięcej wysiłku

JAVATECH

Konwertery

- Powinny rzucać IllegalArgumentException, jeśli nie uda się skonwertować obiektu
- Powinny być bezpieczne ze względu na wątki
- Stanowią alternatywę dla PropertyEditor
- Mogą być używane "ręcznie", programowo

Konwertery typów w Spring 3 – przykład 1

```
final class StringToInteger implements Converter<String, Integer> {

    public Integer convert(String source) {
        return Integer.valueOf(source);
    }
}
```

Konwertery w Spring 3 – przykład 2

```
final class StringToEnumConverterFactory implements ConverterFactory<String, Enum> {

   public <T extends Enum> Converter<String, T> getConverter(Class<T> targetType) {
      return new StringToEnumConverter(targetType);
   }

   private final class StringToEnumConverter<T extends Enum>
         implements Converter<String, T> {

      private Class<T> enumType;

      public StringToEnumConverter(Class<T> enumType) {
         this.enumType = enumType;
      }

      public T convert(String source) {
         return (T) Enum.valueOf(this.enumType, source.trim());
      }
   }
}
```

Rejestracja i dostęp do konwerterów

- Rejestracja:

```
<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean">
  <property name="converters">
    <list>
      <bean class="example.ExoticTypeConverter"/>
    </list>
  </property>
</bean>
```

- API conversionService:
  - boolean canConvert(Class<?> sourceType, Class<?> targetType)
  - boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType)
  - <T> T convert(Object source, Class<T> targetType)
  - Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType)

JAVATECH

Formatowanie pól

- Interfejs Formatter<T>
- Metody:
  - String print(T obj, Locale l) – zwraca reprezentację tekstową dla danej lokalizacji
  - T parse(String str, Locale l) – odtwarza obiekt z reprezentacji tekstowej dla danej lokalizacji

Wbudowane Formattery

- NumberFormatter
- CurrencyFormatter
- PercentFormatter
- DateFormatter

JAVATECH

Formatowanie za pomocą adnotacji

- @DateTimeFormat – wymaga biblioteki JodaTime na classpath
- @NumberFormat
- własne adnotacje – za pomocą implementacji interfejsu AnnotationFormatterFactory<AnnotationClass>
- Przykład użycia adnotacji:

```
public class Person {

  private String name;

  @DateTimeFormat(pattern = "yyyy-MM-dd")
  private Date birthDate;

}
```

Rejestrowanie Formatterów

- Utworzyć klasę pochodną do FormattingConversionServiceFactoryBean
- Przedefiniować metodę installFormatters:

```
public class CustomFormattingConversionServiceFactoryBean
extends          FormattingConversionServiceFactoryBean {

   @Override
    protected void installFormatters(FormatterRegistry registry) {
       // instalacja domyślnych formatterów:
       super.installFormatters(registry);

       // instalacja formattera dla danego typu pola:
       registry.addFormatterForFieldType(
          clazz, new MyCustomFormatter());

       // instalacja formattera konfigurowanego adnotacją:
       registry.addFormatterForFieldAnnotation(
          new MyCustomAnnotationFormatterFactory());
    }
}
```

Programowe użycie walidatorów, konwerterów i formatterów

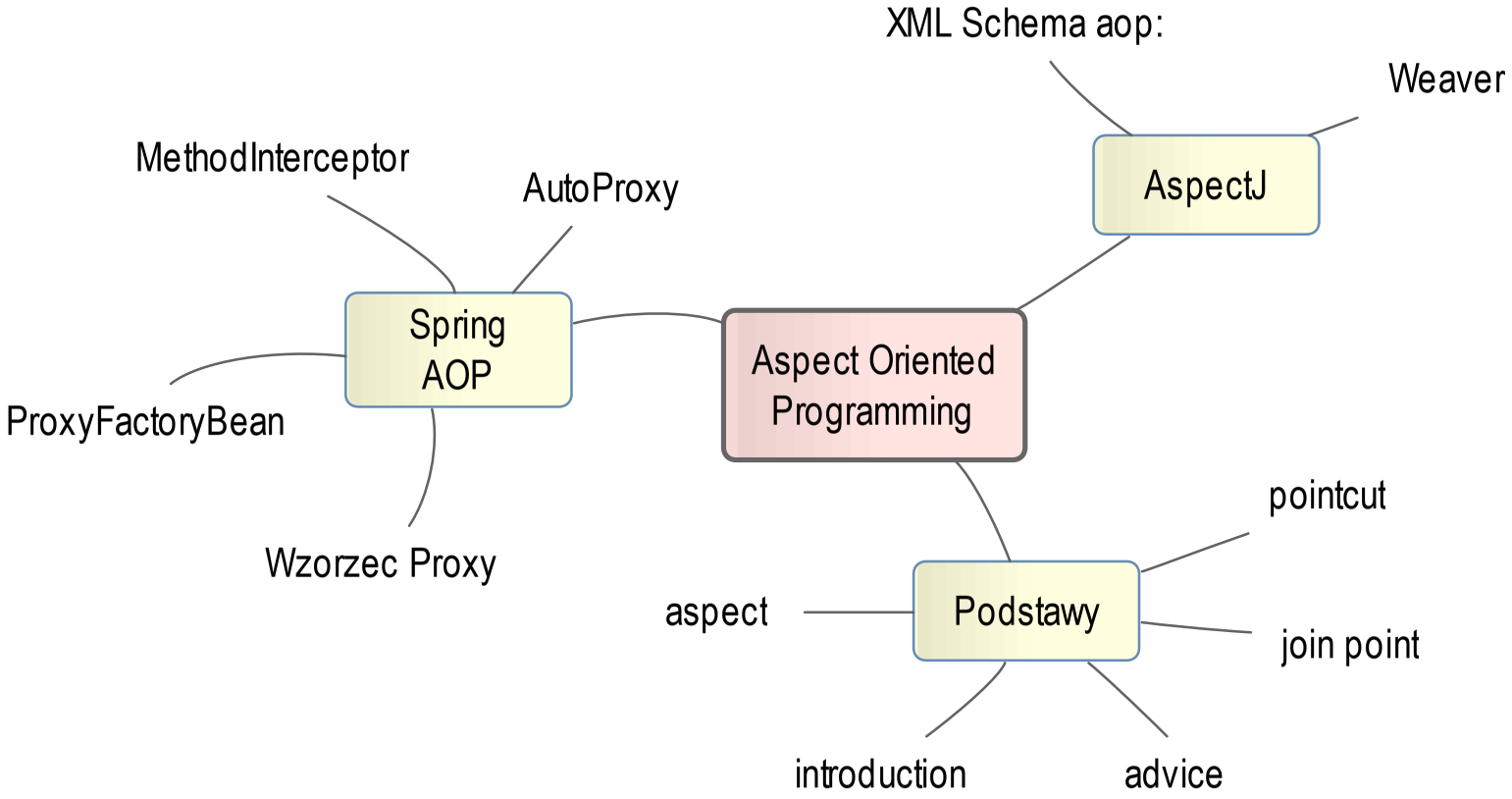
```
@Autowired
private Validator validator;

@Autowired
private ConversionService conversionService;


public void businessMethod() {
    MutablePropertyValues pv = new MutablePropertyValues();
    pv.add("birthDate", "1980-02-06");
    pv.add("name", "John Smith");

    Person person = new Person();
    DataBinder binder = new DataBinder(person);
    binder.setConversionService(conversionService);
    binder.setValidator(validator);
    binder.bind(pv);

            // Walidacja i wyciągnięcie informacji o błędach:
    binder.validate();
    BindingResult br = binder.getBindingResult();

            if (!br.hasErrors())
        // ... tu można używać obiektu person
}
```

JAVATECH

Spring AOP

Programowanie aspektowe
z użyciem Spring AOP
i AspectJ

Plan


XML Schema aop:
Weaver
AspectJ
MethodInterceptor
AutoProxy
Spring AOP
ProxyFactoryBean
Wzorzec Proxy
Aspect Oriented Programming
pointcut
aspect
Podstawy
join point
introduction
advice

JAVATECH

AOP

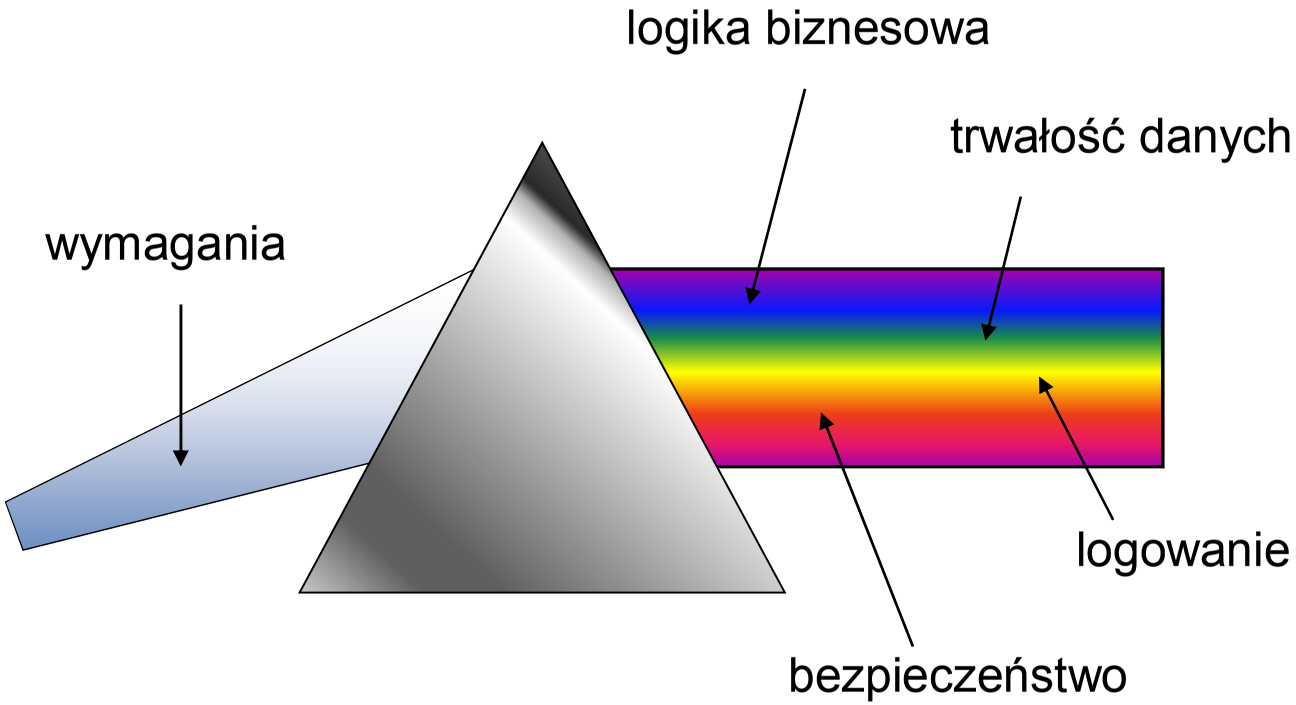
- Aspect Oriented Programming – programowanie aspektowe – próba rozwiązania ograniczeń pojawiających się przy stosowaniu analizy, projektowania i programowania zorientowanego obiektowo do złożonych problemów.
- Programowanie zorientowane aspektowo wraz z koncepcją komputerowej refleksji należy do metod separacji zagadnień (ang. separation of concerns).

JAVATECH

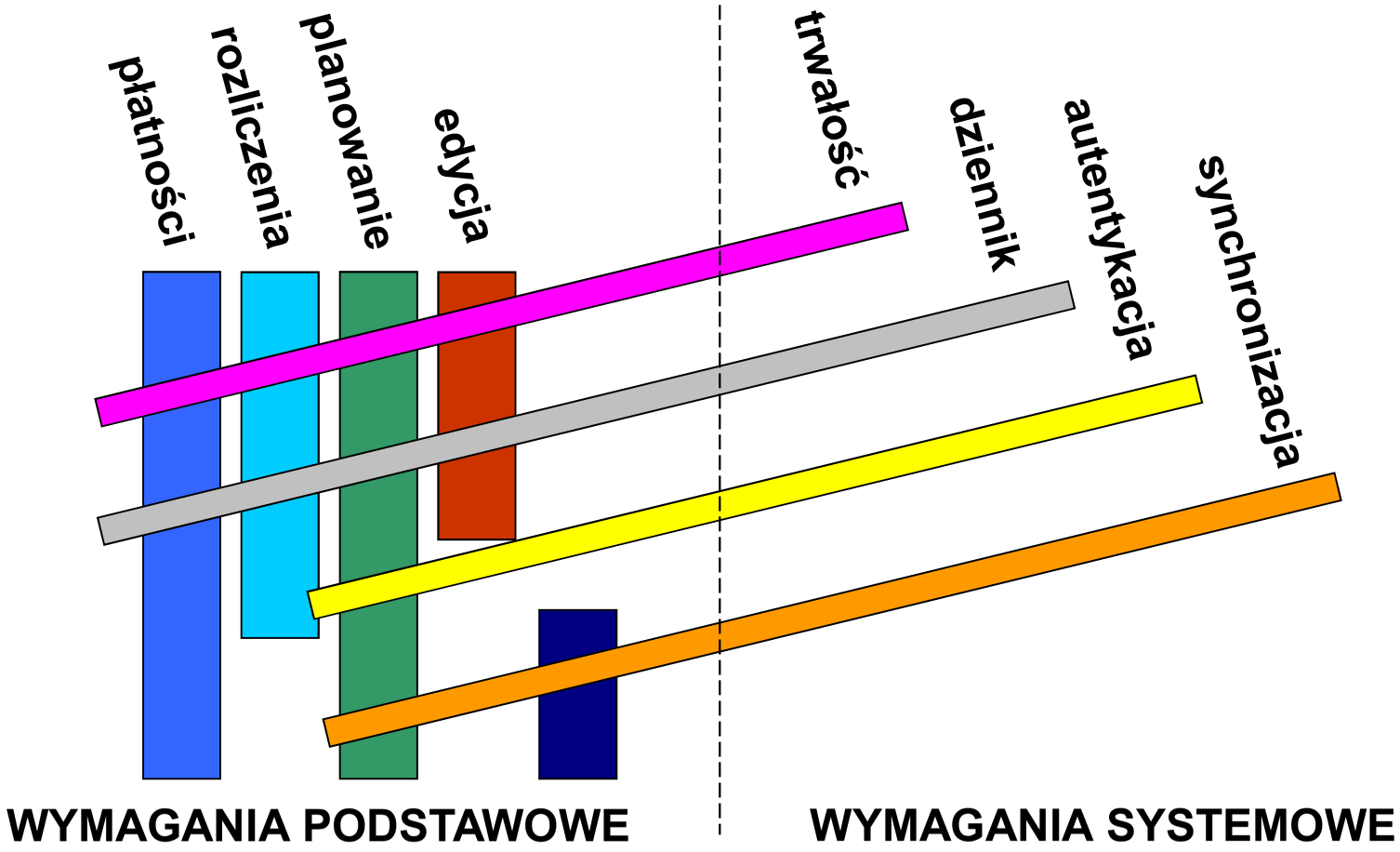
Modularyzacja

- Modularyzacja jest podstawą prawidłowej pielęgnacji kodu, pisali najwięksi badacze inżynierii oprogramowania. Postulowali oni dekompozycję programu na części, z których każda będzie dotyczyła jednego zagadnienia.
- Kolejne paradygmaty programowania, począwszy od programowania funkcyjnego, poprzez strukturalne, aż po obiektowe, próbowały spełnić ten postulat.
- W kolejnych generacjach języków i metod programowania pojawiały się nowe koncepcje, które dzieliły program według różnych kryteriów.

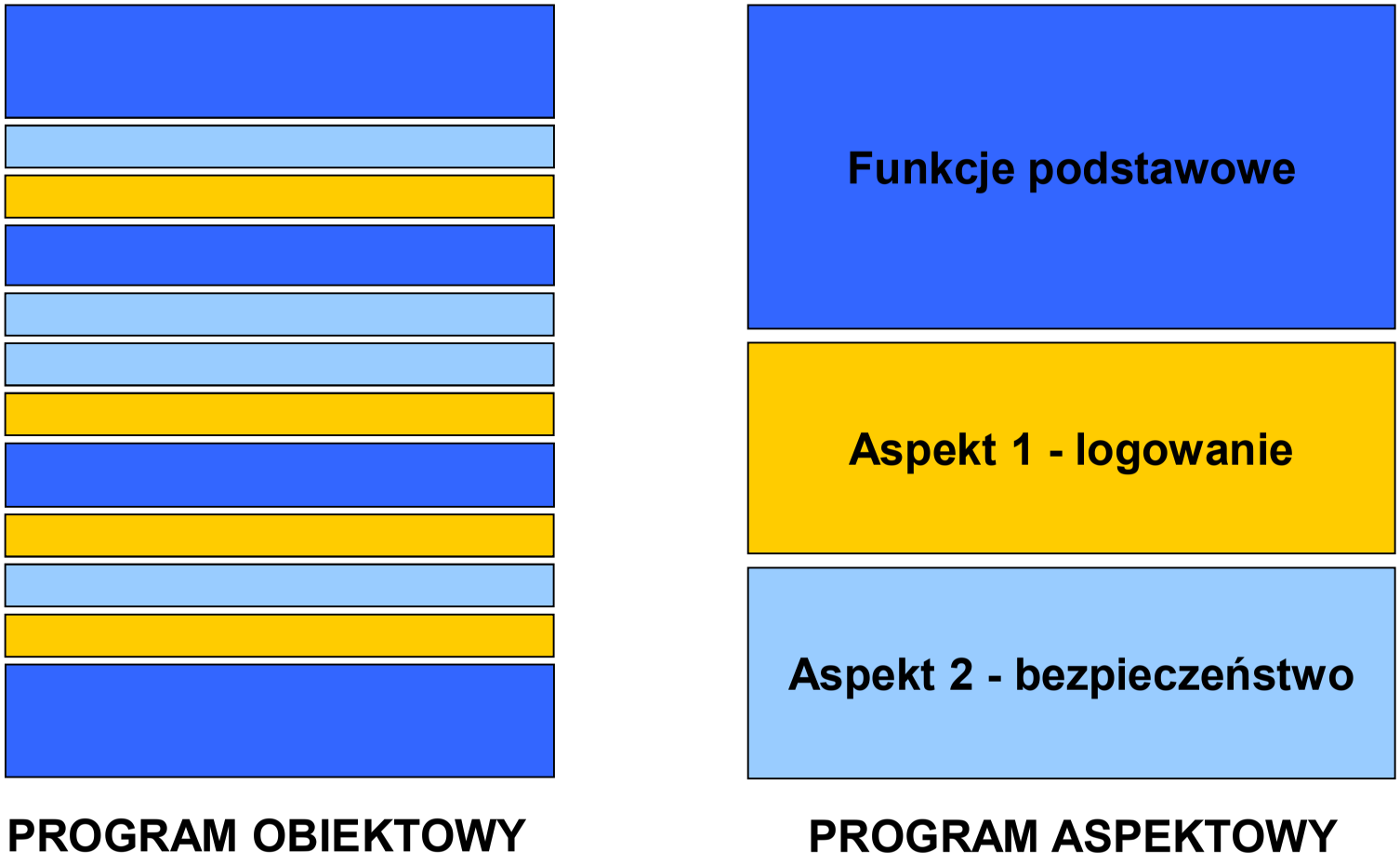
Wymagania klienta



Podział wymagań



Modularyzacja kodu



JAVATECH

OOP a AOP

- Programowanie obiektowe
  - grupowanie podobnych koncepcji za pomocą hermetyzacji i dziedziczenia
  - podstawowa jednostka modularyzacji: klasa

- Programowanie aspektowe
  - grupowanie podobnych koncepcji w niezwiązanych ze sobą klasach
  - dodatkowy mechanizm modularyzacji: aspekt

Podstawowe pojęcia AOP

- Punkty złączenia (ang. join point)
- Punkt cięcia (ang. pointcut)
- Porada (ang. advice)
- Wprowadzenie (ang. introduction)
- Aspekt (ang. aspect)

Punkty złączenia

- **Punkty złączenia** (ang. joinpoints) są dowolnymi, identyfikowalnymi miejscami w programie, posiadają własny kontekst:
  - wywołanie metody i konstruktora
  - wykonanie metody i konstruktora
  - dostęp do pola
  - obsługa wyjątku
  - statyczna inicjacja

JAVATECH

Punkt cięcia

- **Punkt cięcia** (ang. *pointcut*) jest zdefiniowaną kolekcją punktów złączenia. Ma dostęp do ich kontekstu.

JAVATECH

Porada

- **Porada** (ang. *advice*) jest fragmentem kodu programu wykonywanym przed, po lub zamiast osiągnięcia przez program punktu cięcia.

JAVATECH

Wprowadzenie

- **Wprowadzenie** (ang. introduction) to proces, który umożliwia modyfikację struktury obiektu przez wprowadzenie dodatkowych metod lub pól.

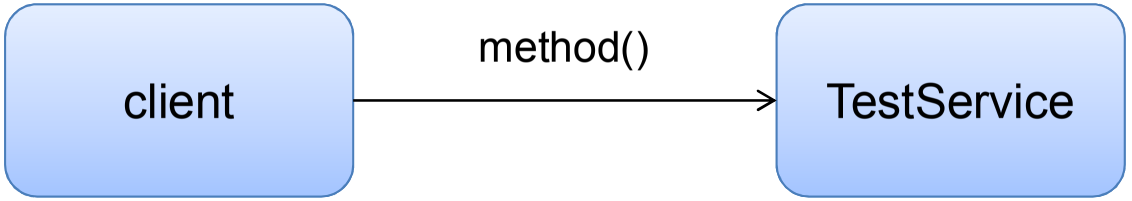
Aspekt

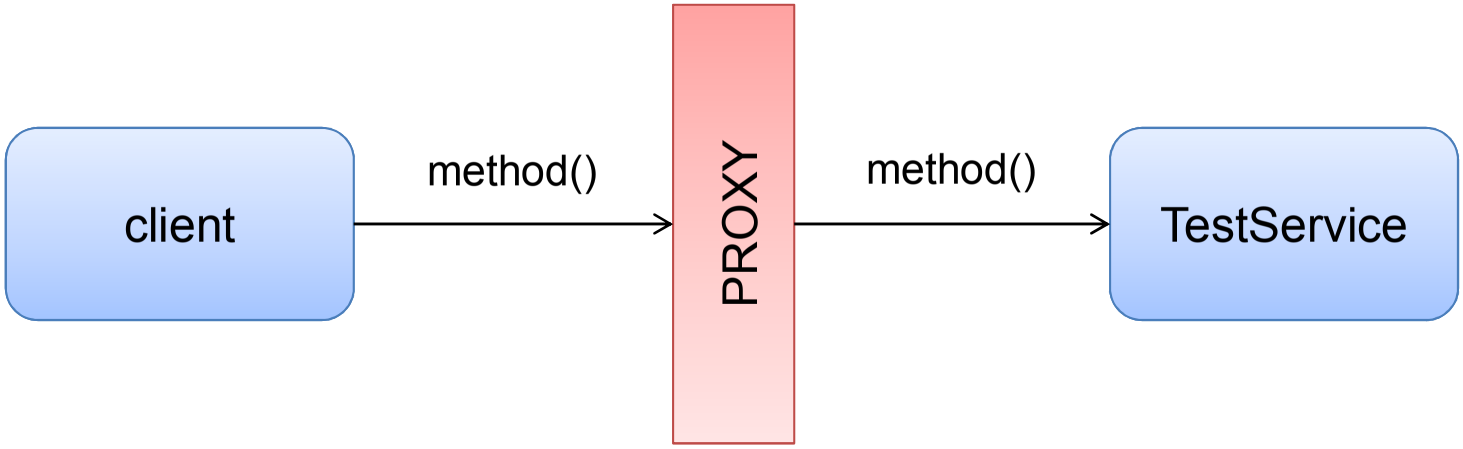
- **Aspekt** (ang. aspect) to kombinacja porad i punktów cięcia. Definiuje jaka logika ma zostać dołączona do aplikacji i gdzie ma zostać ona wywołana.

Spring AOP

- Spring umożliwia stosowanie programowania aspektowego.
- Spring posiada własne AOP zaimplementowaną w oparciu o mechanizmy *proxy*.
- Spring wspiera również AspectJ.

Koncepcja Proxy





Koncepcja Proxy (cd)

```
public interface TestService {
  public String getData();
  public void setData(String data);
}

public class TestServiceImpl implements TestService {

  public String getData() {
    return "data";
  }

  public void setData(String data) {

  }

}
```

Wywołanie usługi

```
public static void main(String[] args) {
  TestService service = new TestServiceImpl();
  service.setData("DATA");
}


public static void main(String[] args) {
  ApplicationContext context =
        new ClassPathXmlApplicationContext("/context.xml");
  TestService service = (TestService) context.getBean("service");
  service.setData("DATA");
}
```

JDK Proxy

```
class LoggingInvocationHandler implements InvocationHandler {

  public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        ...
        return method.invoke(target, args);
  }



InvocationHandler handler = new LoggingInvocationHandler();
TestService serviceProxy = (TestService) Proxy.newProxyInstance(
        Main.class.getClassLoader(), new Class[] { TestService.class },
                handler);
serviceProxy.setData("DATA");
```

JAVATECH

Spring Proxy

```
public class Main {

  public static void main(String[] args) {
    ProxyFactory factory = new ProxyFactory(new TestServiceImpl());
    factory.addInterface(TestService.class);
    factory.addAdvice(new BeforeInterceptor());
    TestService service = (TestService) factory.getProxy();
    service.setData("DATA");
  }

}
```

Spring Proxy w XML

```
<bean id="serviceBean" class="test.TestServiceImpl"/>
<bean id="before" class="test.BeforeInterceptor"/>

<bean id="service"
      class="org.springframework.aop.framework.ProxyFactoryBean">
 <property name="proxyInterfaces" value="test.TestService"/>
 <property name="target" ref="serviceBean"/>
 <property name="interceptorNames">
  <list>
   <value>before</value>
  </list>
 </property>
</bean>
```

Porady w Spring

- org.springframework.aop.MethodBeforeAdvice
- org.springframework.aop.ThrowsAdvice
- org.springframework.aop.AfterReturningAdvice
- org.aopalliance.intercept.MethodInterceptor

org.springframework.aop.MethodBeforeAdvice

```
public class BeforeInterceptor implements MethodBeforeAdvice {

    public void before(Method method, Object[] args, Object target)
     throws Throwable {
        ...
        ...
        ...
    }

}
```

org.springframework.aop.AfterReturningAdvice

```
public class AfterReturningInterceptor implements
        AfterReturningAdvice {

  @Override
  public void afterReturning(Object returnValue, Method method,
        Object[] arguments, Object target) throws Throwable {

  }

}
```

org.aopalliance.intercept.MethodInterceptor

```
public class InvokeInterceptor implements MethodInterceptor {

  @Override
  public Object invoke(MethodInvocation invocation) throws Throwable {

  return null;
  }

}
```

AutoProxying

```
<bean id="preformanceThresholdProxyCreator"
      class="org.springframework.aop.framework.
            autoproxy.BeanNameAutoProxyCreator">
  <bean>
    <property name="beanNames">
      <list>
        <value>*Service</value>
      </list>
    </property>
    <property name="interceptorNames">
      <value>debug</value>
      <value>performance</value>
    </property>
  </bean>
<bean id="debug"
        class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="performance"
        class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>
```

DefaultAdvisorAutoProxyCreator

```
<bean id="advisor" class="org.springframework.aop.support.
                          RegexpMethodPointcutAdvisor">
    <property name="advice">
      <bean class="performanceThresholdInterceptor"/>
    </property>
    <property name="pattern">
      <value>.+Service\..+</value>
    </property>
  </bean>

  <bean id="autoProxyCreator"
     class="org.springframework.aop.framework.
                    autoproxy.DefaultAdvisorAutoProxyCreator"/>
```

JAVATECH

AspectJ

- G. Kiczales (2001), Xerox Palo Alto Research Center
- uniwersalne aspektowe rozszerzenie Javy
- aspekt jako specyficzna klasa
- możliwość zmiany zachowania i struktury kodu
- łączenie aspektów i klas na poziomie bajtkodu
- własny kompilator *ajc*
- integracja z Eclipse IDE

Schema based AOP

- Spring dostarcza wygodnego mechanizmu definiowania AOP za pomocą dodatkowego schematu xsd.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop.xsd">
```

Deklaracja aspektu

```
<aop:config>
        <aop:aspect id="myAspect" ref="aBean">
        ...
        </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
...
</bean>
```

Deklaracja pointcut

```
<aop:config>
<aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>
</aop:config>
```

Deklaracja advice

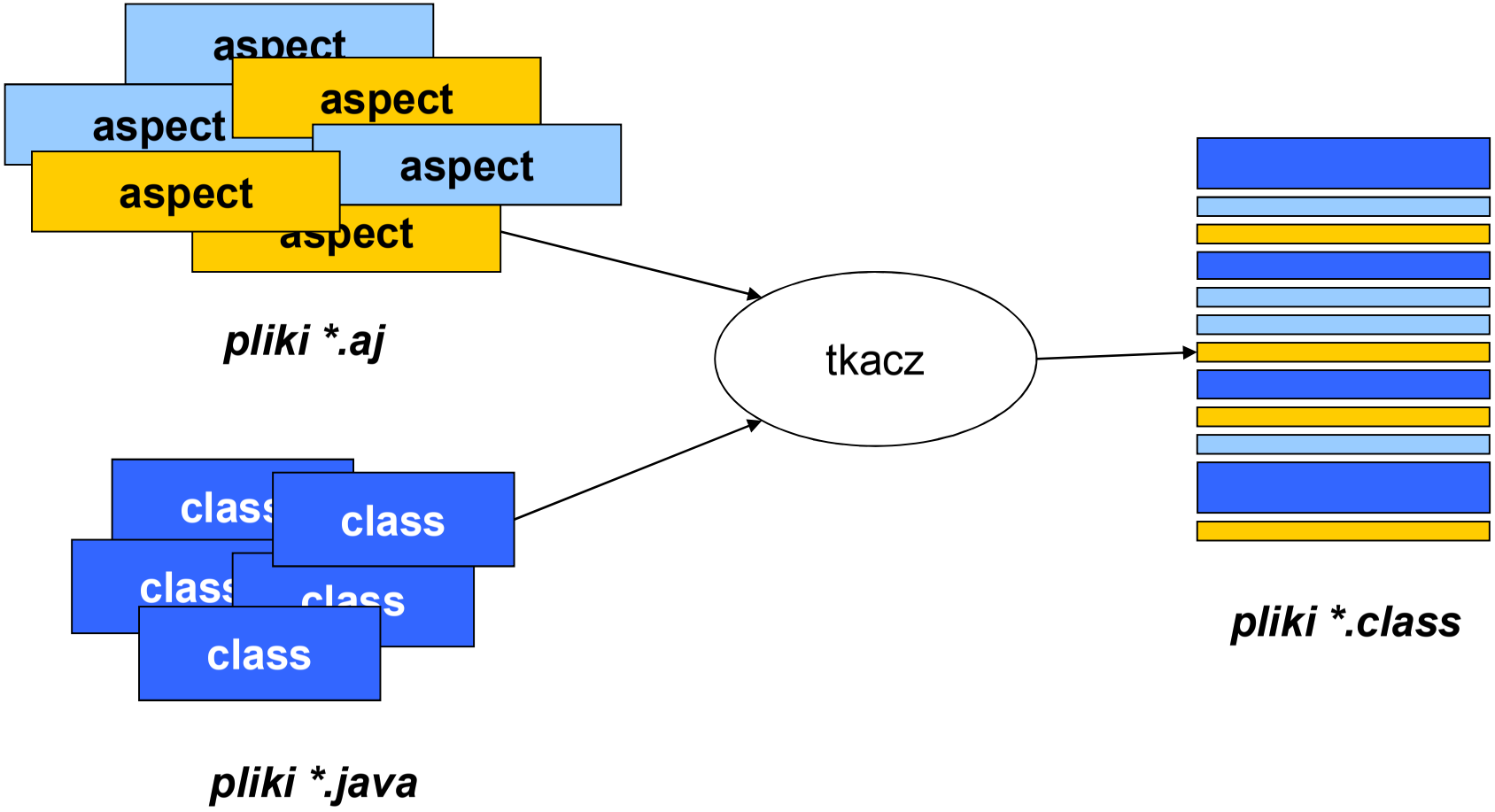
```
<aop:before pointcut-ref="businessService" method="myMethod"/>

<aop:after/>

<aop:after-returning/>

<aop:after-throwing/>

<aop:around/>
```

Przykład

```
public class OrderedMonitorAdvice{
        public Object operation(ProceedingJoinPoint pjp) throws Throwable {
        …
        …
}
```

```
<bean id="orderedMonitorAdvice" class="OrderedMonitorAdvice">
        <property name="…" value="…"/>
</bean>

<aop:config>
<aop:aspect id="orderedMonitor" ref="orderedMonitorAdvice">
<aop:pointcut id="operation"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>
<aop:around pointcut-ref="operation"
method="operation"/>
```

Weaver (AspectJ)



JAVATECH

Wykorzystanie AspectJ

- Korzystanie z AspectJ wymaga kompilacji aplikacji za pomocą kompilatora ajc lub użycie tzw. load-time weaving czyli kompilacji aspektów w czasie ładowania klasy do jvm.

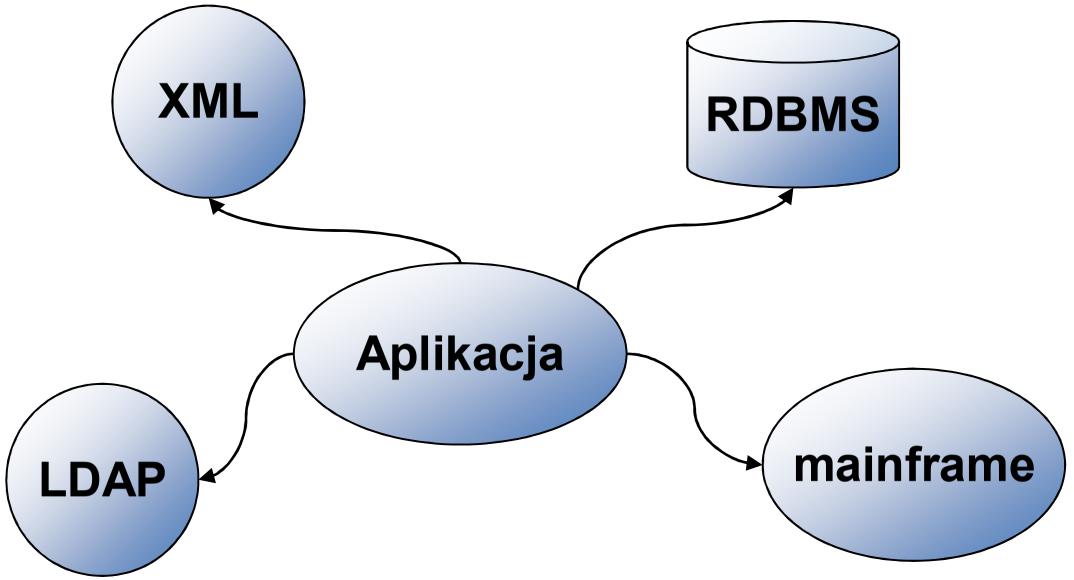
JAVATECH

Spring JPA

Wsparcie dla warstwy DAO
w Spring Framework

Problem

- Większość aplikacji biznesowych jako trwałych magazynów używa systemów zarządzania relacyjnymi bazami danych (RDBMS). Jednak dane biznesowe mogą znajdować się również w innych miejscach np zewnętrznych systemach mainframe, repozytoriach LDAP, obiektowych bazach danych, plikach.



DAO

- Data Access Object
  - wzorzec projektowy umożliwiający oddzielenie warstwy implementacji dostępu do danych od aplikacji
  - zyskujemy możliwość korzystania z różnych rodzajów źródeł danych bez konieczności zmian w aplikacji

DAO w Spring

- JdbcDaoSupport
- HibernateDaoSupport
- JdoDaoSupport
- JpaDaoSupport
- SqlMapClientDaoSupport

JAVATECH

PersonDao

```
public interface PersonDao {

    public Person get(Long id);

    public List<Person> selectAll();

    public void save(Person person);

    public void delete(Person person);

}
```

PersonDaoImpl

```
public class PersonDaoImpl
        implements PersonDao {

    @Override
    public Person get(Long id) {
    …
    }

    @Override
    public void delete(Person person) {
    …
    }

 …
}
```

Konfiguracja DataSource w Spring

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

<!-- alternatywnie -->

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

LocalEntityManagerFactoryBean

- Najprostrza wersja posiadająca jednak szereg ograniczeń np. brak możliwości odwołania się do komponentu DataSource czy też integracji z globalnymi transakcjami.
- Dobra opcja dla małych aplikacji standalone lub działającej poza serwerem aplikacji wspierającym JPA oraz do testów integracyjnych.

```
<bean id="myEmf"
        class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="myPersistenceUnit"/>
</bean>
```

Pozyskanie EntityManagerFactory z JNDI

- Wersja dobra dla aplikacji pracującej pod kontrolą serwera aplikacyjnego.
- Integruje się z zarządcą transakcji JTA.
- Umożliwia współdzielenie kontekstu JPA pomiędzy aplikacjami.

```
<jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersistenceUnit"/>
```

LocalContainerEntityManagerFactory Bean

- Najbardziej zaawansowana wersja pozwalająca na zdefiniowanie wszystkich aspektów konfiguracji JPA na poziomie kontenera Spring.

```
<bean id="myEmf"
class="org.springframework.orm.jpa. LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource"/>
</bean>
```

Pozyskanie EMF w aplikacji

```
public class PersonDaoImpl
        implements PersonDao {


  private EntityManagerFactory emf;

  public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.emf = emf;
  }


  …
}
```

EMF można pozyskać w tradycyjny sposób wstrzykując setterem komponent z kontenera.

Pozyskanie EMF w aplikacji

```
public class PersonDaoImpl
        implements PersonDao {


  private EntityManagerFactory emf;

  @PersistenceUnit
  public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.emf = emf;
  }

   …
}
```

```
<bean
class="org.springframework.orm.jpa.support
        .PersistenceAnnotationBeanPostProcessor"/>
<!-- lub -->
<context:annotation-config/>
```

EMF można pozyskać również za pomocą adnotacji.
Wymaga to dodatkowej konfiguracji kontenera.

OpenEntityManagerInView

- Ze względu na charakterystykę działania aplikacji web istnieje prawdopodobieństwo wystąpienia problemów z lazy loading w JPA.
- W takim przypadku zalecane jest użycie filtru wprowadzającego do aplikacji mechanizm utrzymywania otwartej sesji w ramach całego requestu.

JAVATECH

Spring TX

Mechanizmy transakcji
w środowisku Spring Framework

JAVATECH

Transakcja

- Grupa operacji widziana jako pojedyncza operacja.
- W transakcji dochodzi do wykonania wszystkich operacji albo żadnej z nich.
- Operacje wykonywane w ramach jednej transakcji mogą działać na różnych serwerach i źródłach danych.

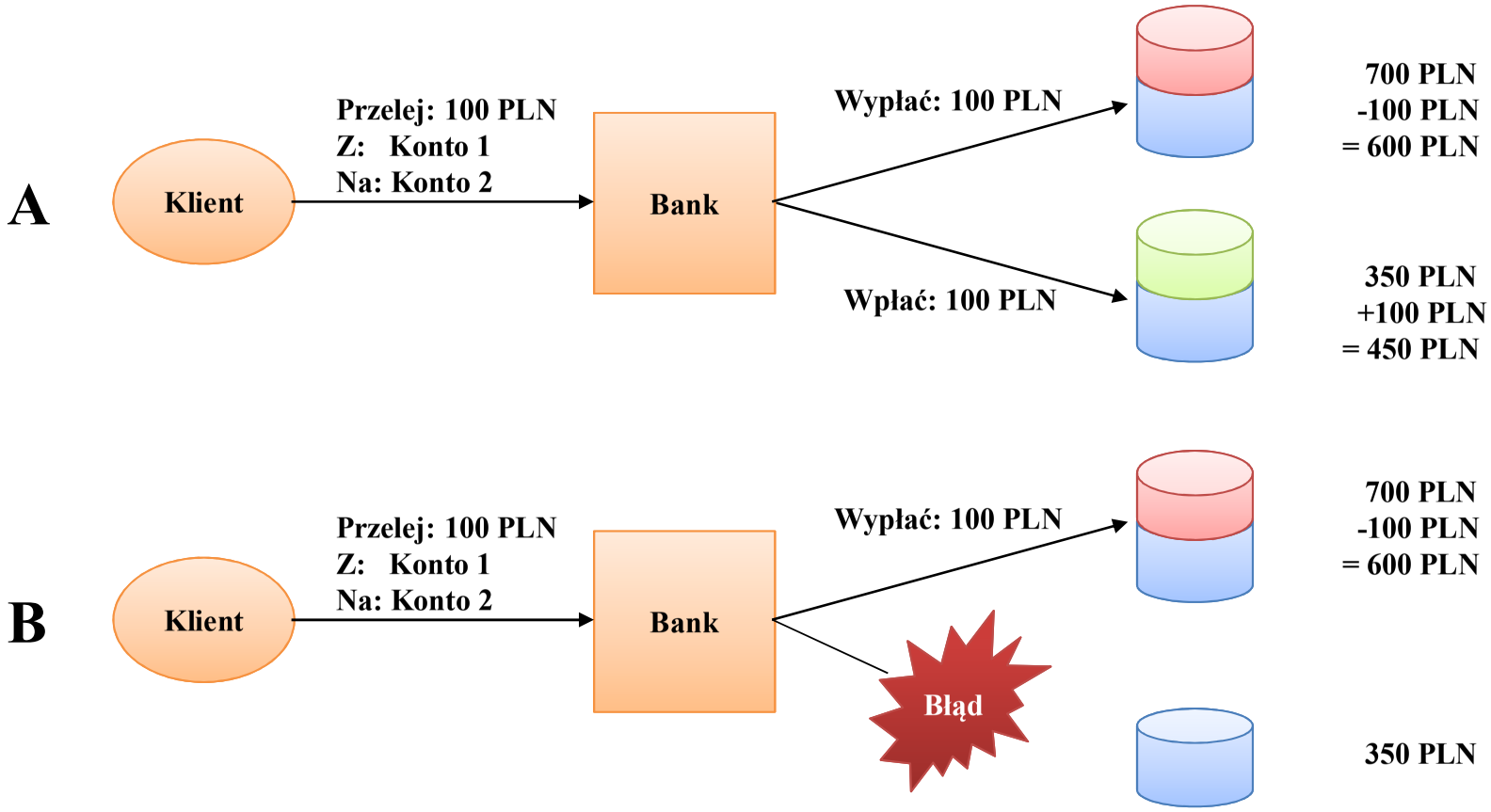
JAVATECH

ACID

- Atomicity: wszystkie operacje wchodzące w skład transakcji zostają wykonane albo żadna z nich nie zostaje wykonana.
- Consistency: po zakończeniu transakcji system musi znajdować się w stabilnym i spójnym stanie
- Isolation: transakcje odbywają się niezależnie od innych operacji (modyfikacje wykonane przez operacje wchodzące w skład transakcji nie są widziane poza nią do czasu zakończenia)
- Durablility: zakończone transakcje są trwałe (istnieje możliwość odtworzenia stanu po transakcji nawet po uszkodzeniu systemu)

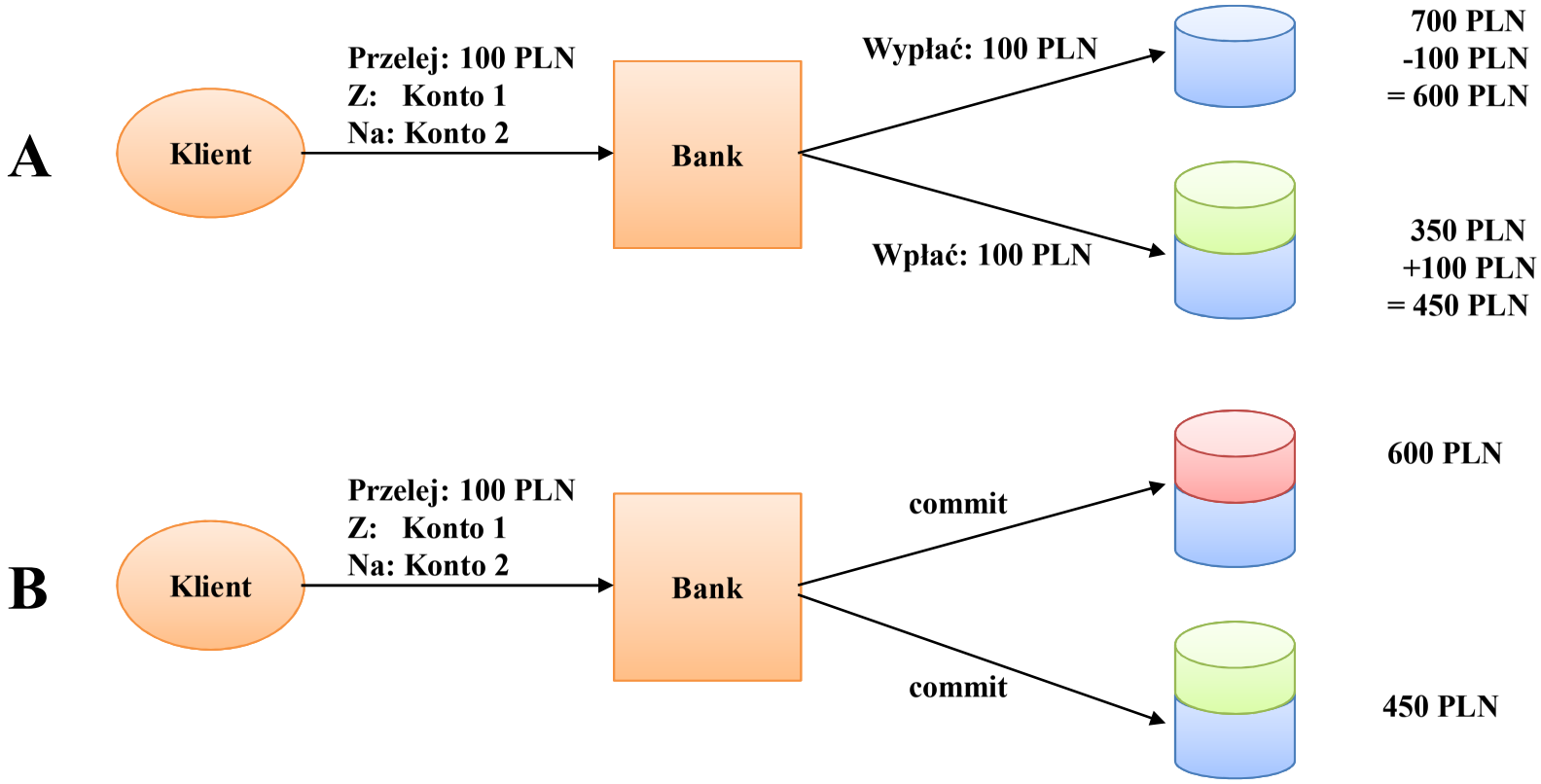
Transakcje lokalne

- Przykład: transfer z jednego konta na drugie bez transakcji (A – transfer udany, B- transfer nieudany)



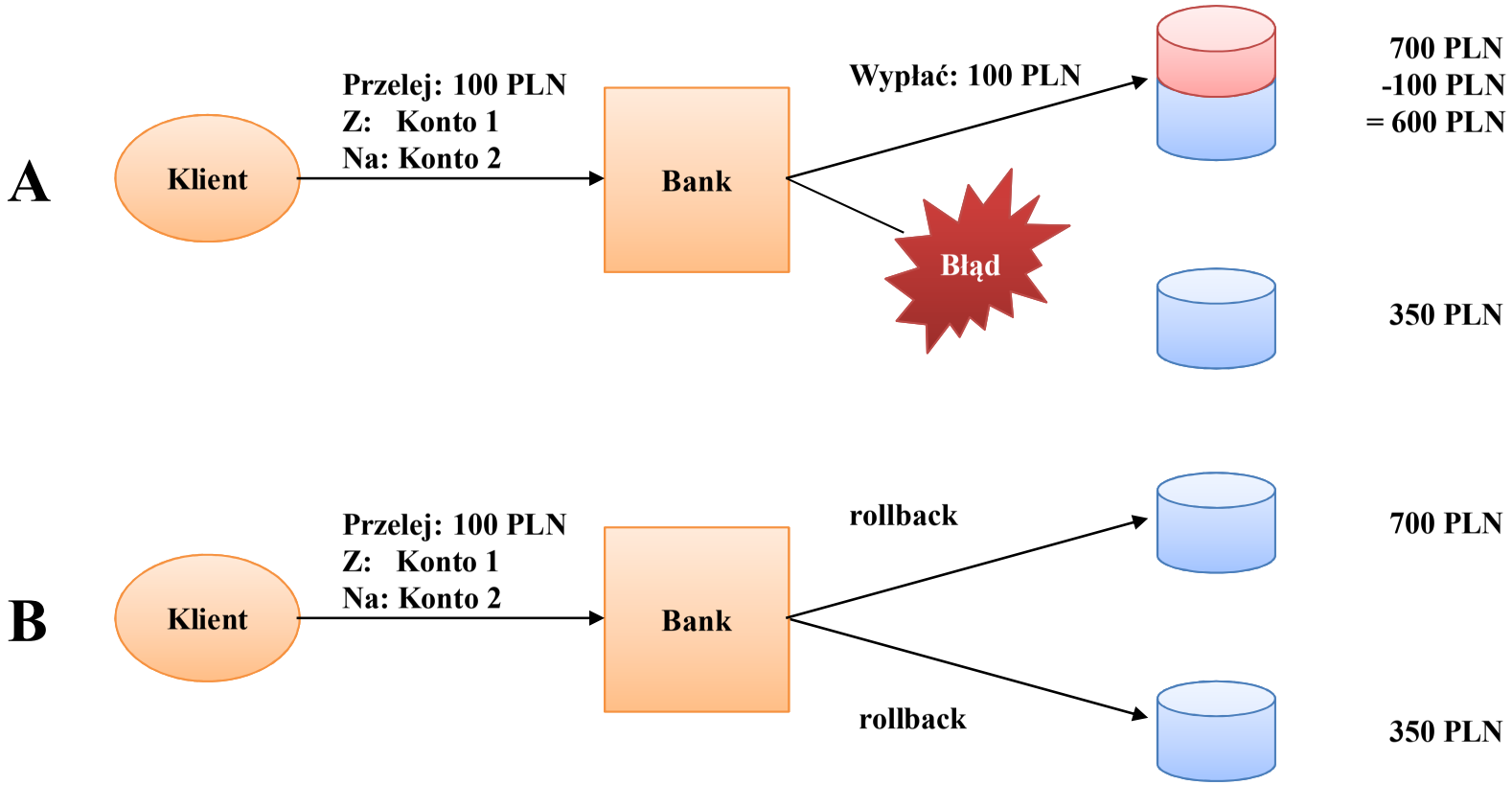
Transakcje lokalne

- Przykład: transfer z jednego konta na drugie z transakcją (A – faza pierwsza, B- potwierdzenie)



Transakcje lokalne

- Przykład: transfer z jednego konta na drugie z transakcją (A – faza pierwsza, B- wycofanie)



JAVATECH

Obsługa transakcji w Spring

- Spring nie wprowadza własnych mechanizmów obsługi transakcji.
- Spring nie obsługuje transakcji bezpośrednio a jedynie za pomocą klas zarządzających transakcjami deleguje do odpowiednich mechanizmów charakterystycznych dla danej platformy (JTA, JDBC, Hibernate itp.).

Zarządcy transakcji

org.springframework.jdbc.datasource. **DataSourceTransactionManager**	Zarządza transakcjami na pojedynczym obiekcie JBDC DataSource
org.springframework.orm.hibernate. **HibernateTransactionManager**	Zarządza transakcjami w przypadku użycia Hibernate jako mechanizmu pesystencji.
org.springframework.orm.jdo. **JdoTransactionManager**	Zarządza transakcjami w przypadku użycia JDO jako mechanizmu pesystencji.
org.springframework.orm.jpa. **JpaTransactionManager**	Zarządza transakcjami w przypadku użycia JPA jako mechanizmu pesystencji.
org.springframework.transaction. **jta.JtaTransactionManager**	Zarządza transakcjami z użyciem Java Transaction API np. w środowiskach zarządzanych.
org.springframework.orm.ojb. **PersistenceBrokerTransactionManager**	Zarządza transakcjami w przypadku użycia Apache OJB jako mechanizmu pesystencji.

PersonService

```
public interface PersonService{
        public void deletePersons(List<Person> Persons);
}

public class PersonServiceImpl implements PersonService{

    private TransactionTemplate transactionTemplate;

    public void setTransactionTemplate(TransactionTemplate transactionTemplate) {
        this.transactionTemplate = transactionTemplate;
    }

    @Override
    public void deletePersons(final List<Person> Persons) {


    }
}
```

Użycie TransacionTemplate

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

<bean id="transactionTemplate"
class="org.springframework.transaction.support.TransactionTemplate">
        <property name="transactionManager" ref="transactionManager"/>
</bean>

<bean id="PersonService" class="service.PersonServiceImpl">
        <property name="transactionTemplate" ref="transactionTemplate" />
</bean>
```

Określenie transakcji

```
@Override
public void deletePersons(final List<Person> Persons) {

        transactionTemplate.execute(new TransactionCallback() {

        @Override
        public Object doInTransaction(TransactionStatus status) {
                ...
                ...
        }

}
```

Transakcje deklaratywne

- Transakcje określane w kodzie dają dużą kontrolę nad jej granicami jednak zmiany mogą być nieco uciążliwe.
- Alternatywą dającą równie dużą kontrolę ale większą elastyczność są transakcje deklaratywne czyli określane za pomocą adnotacji lub plików konfiguracyjnych.

TransactionProxyFactoryBean

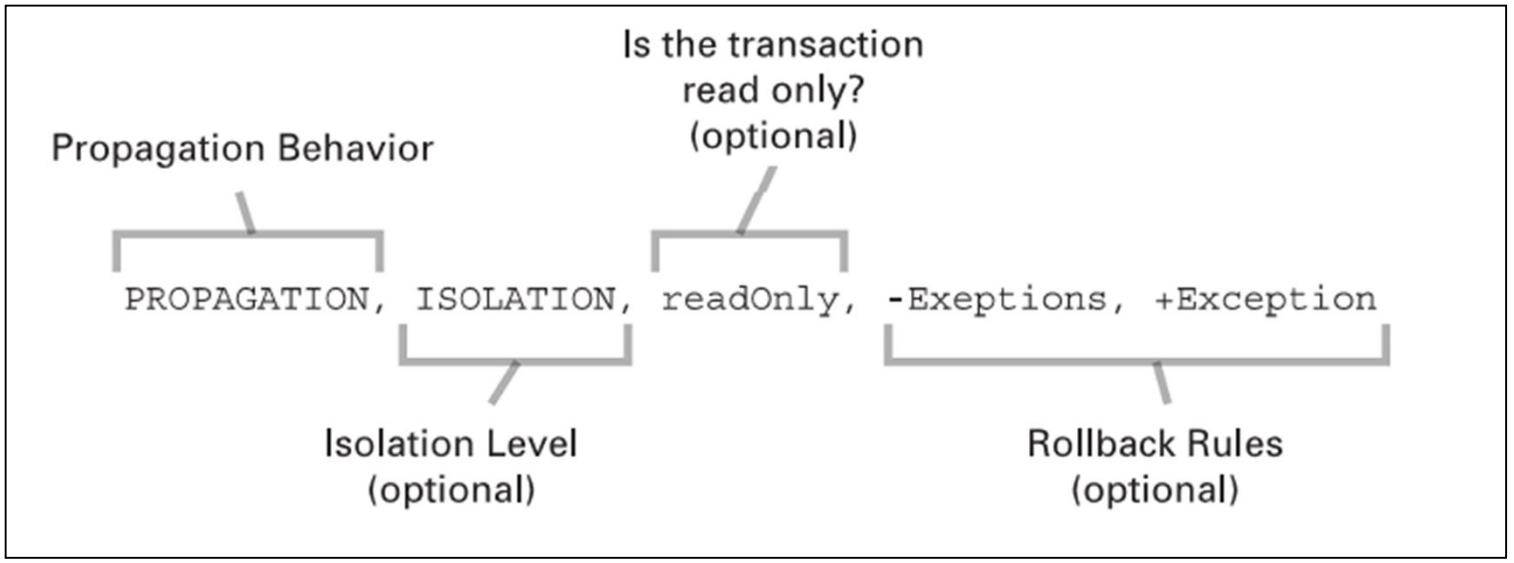
```
<bean id="PersonService"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="target">
                <bean class="service.PersonServiceImpl">
                <property name="PersonDao" ref="PersonDao" />
                </bean>
        </property>
        <property name="transactionAttributes">
        <value>
                *=PROPAGATION_REQUIRED
        </value>
        </property>
        <property name="transactionManager" ref="transactionManager" />
</bean>
```

TransactionProxyFactoryBean

```
<bean id="PersonService"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="target">
                <bean class="service.PersonServiceImpl">
                <property name="PersonDao" ref="PersonDao" />
                </bean>
        </property>
        <property name="transactionAttributes">
        <value>
                *=PROPAGATION_REQUIRED
        </value>
        </property>
        <property name="transactionManager" ref="transactionManager" />
</bean>
```

Atrybuty transakcji

- Propagacja transakcji
- Poziom izolacji
- Atrybuty read only
- Timeout



Propagacja transakcji

- PROPAGATION_MANDATORY
- PROPAGATION_NESTED
- PROPAGATION_NEVER
- PROPAGATION_NOT_SUPPORTED
- PROPAGATION_REQUIRED
- PROPAGATION_REQUIRES_NEW
- PROPAGATION_SUPPORTS

Poziom izolacji transakcji

- ISOLATION_DEFAULT (datastore)
- ISOLATION_READ_UNCOMMITTED
- ISOLATION_READ_COMMITTED
- ISOLATION_REPEATABLE_READ
- ISOLATION_SERIALIZABLE

Konfiguracja serwisu

```
<bean id="PersonService"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="target">
                <bean class="service.PersonServiceImpl">
                <property name="PersonDao" ref="PersonDao" />
                </bean>
        </property>
        <property name="transactionAttributes">
        <value>
                *=PROPAGATION_REQUIRED
        </value>
        </property>
        <property name="transactionManager" ref="transactionManager" />
</bean>
```

TransactionProxyFactoryBean

```
<bean id="PersonService"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="target">
                <bean class="service.PersonServiceImpl">
                        <property name="PersonDao" ref="PersonDao" />
                </bean>
        </property>
        <property name="transactionAttributeSource" ref="transactionAttributeSource"/>
        <property name="transactionManager" ref="transactionManager" />
</bean>


<bean id="transactionAttributeSource"
class="org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource">
        <property name="properties">
                <props>
                        <prop key="deletePersons">
                        PROPAGATION_REQUIRED
                        </prop>
                </props>
        </property>
</bean>
```

JAVATECH

AOP w transakcjach

- Spring umożliwia wygodne definiowanie zasięgu transakcji za pomocą AOP.
- Definicję granic transakcji można zdefiniować za pomocą wpisu w konfiguracji XML bądź adnotacji.

AOP w konfiguracji XML

```
<aop:config>
    <aop:pointcut id="allServiceMethods"
            expression="execution(* service.*.*(..))"/>
    <aop:advisor advice-ref="transactionAdvice"
            pointcut-ref="allServiceMethods"/>
</aop:config>

<tx:advice id="transactionAdvice" transaction-manager="transactionManager">
    <tx:attributes>
      <tx:method
            name="*"
            isolation="READ_COMMITTED"
            propagation="REQUIRED"
            timeout="100"/>
      <tx:method
            name="get*"
            read-only="true"/>
    </tx:attributes>
 </tx:advice>
```

Atrybuty <tx:annotation-driven />

`transactionManager`	referencja do managera transakcji
`mode`	Tryb tworzenia komponentów pomocniczych (advice), domyślnie to *proxy* realizujący tą funkcjonalność z pomocą JDK proxy, drugą możliwością jest *aspectj* definiujący wykorzystanie AspectJ
`order`	kolejność tworzenia aspektu
`proxy-target-class`	jeśli true proxy będzie realizowane na docelowej klasie a nie na implementujących przez nią interfejsów

Transakcje za pomocą adnotacji

```
@Transactional
public void deletePersons(final List<Person> Persons) {

        …
        ...

}
```

```
<bean id="PersonService" class=" service.PersonServiceImpl"/>

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

<aop:aspectj-autoproxy />
```

Atrybuty @Transactional

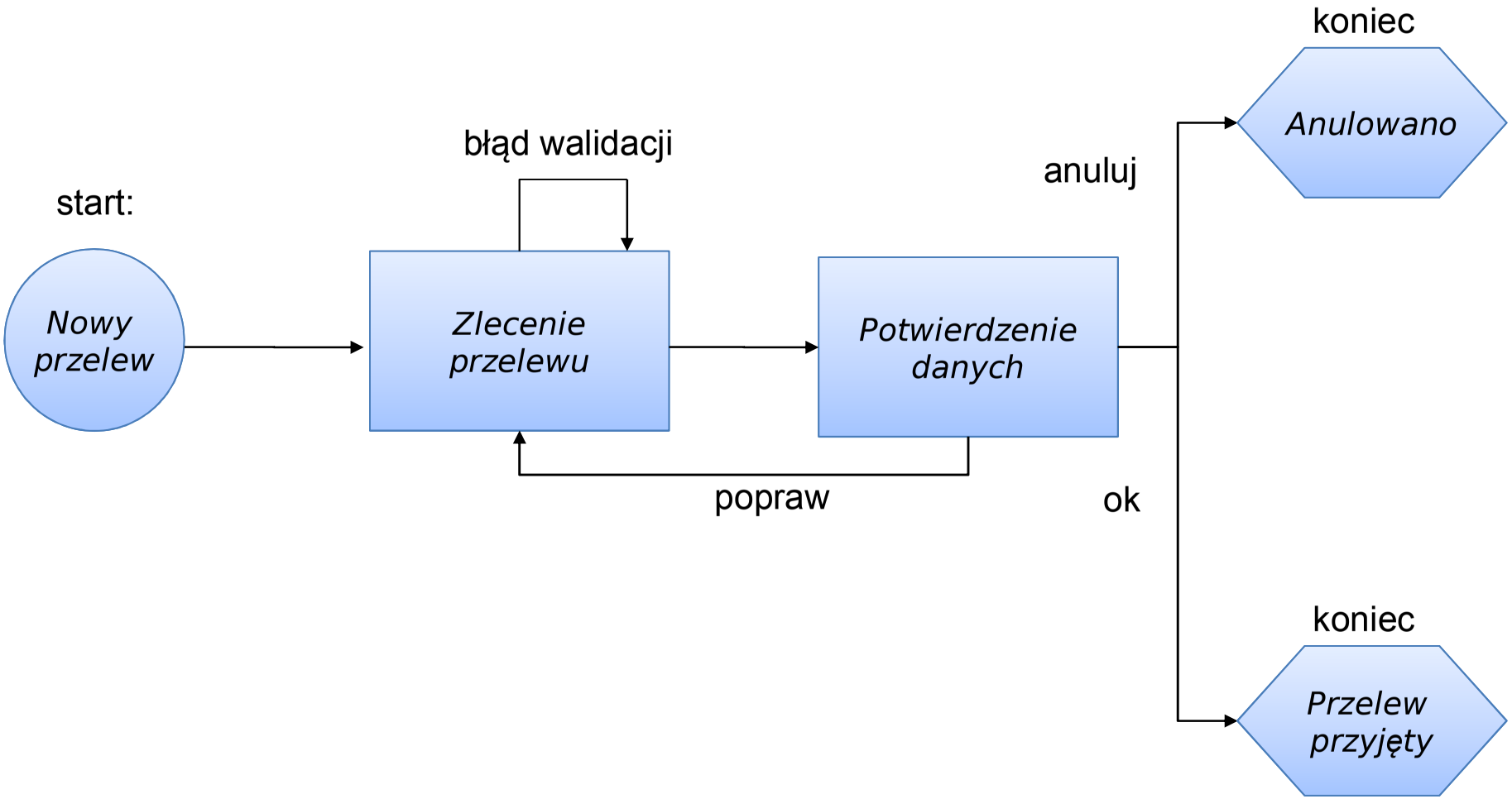
`propagation`	określa sposób propagacji transakcji
`isolation`	określa poziom izolacji transakcji
`timeout`	timeout transakcji w s
`readOnly`	określa czy transakcja jest tylko do odczytu
`noRollbackFor`	tablica klas wyjątków określająca, które z nich mogą zostać wyrzucone przez metodę a które nie mają spowodować wycofania transakcji
`rollbackFor`	tablica klas wyjątków określająca, które z nich mogą zostać wyrzucone przez metodę a które mają spowodować wycofanie transakcji

JAVATECH

Spring WebFlow

Tworzenie aplikacji web

Przepływ


koniec
Anulowano
błąd walidacji
anuluj
start:
Nowy przelew
Zlecenie przelewu
Potwierdzenie danych
popraw
ok
koniec
Przelew przyjęty

Integracja Spring Webflow ze Spring MVC

```
<!-- FlowHandlerAdapter i FlowHandlerMapping są odpowiedzialne za
        przechwycenie żądania
    i skierowanie do odpowiedniego obiektu wykonującego przepływ -->
 <bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
  <property name="flowExecutor" ref="flowExecutor" />
 </bean>
 <bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
        <property name="flowRegistry" ref="flowRegistry"/>
        <property name="order" value="0"/>
 </bean>

 <!-- Tu podajemy listę plików z definicjami przepływów -->
 <webflow:flow-registry id="flowRegistry">
        <webflow:flow-location path="/WEB-INF/flows/transfer.xml" />
 </webflow:flow-registry>

 <!-- Domyślna implementacja zarządcy przepływów -->
 <webflow:flow-executor id="flowExecutor" />

</beans>
```

Integracja c.d.

- Aby WebFlow uwzględniał logiczne nazwy widoków w Spring MVC i używał wskazanego resolvera widoków, należy dopisać:

```
<webflow:flow-builder-services id="flowBuilderServices"
      view-factory-creator="mvcViewFactoryCreator"/>

<bean id="mvcViewFactoryCreator"
     class="org.springframework.webflow.mvc.builder.MvcViewFactoryCreator">
  <property name="viewResolvers" ref="viewResolver"/>  <!-- resolver widoków -->
</bean>
```

JAVATECH

Definiowanie przepływów

- W aplikacji można mieć wiele przepływów
- Definicja przepływu = plik XML zawierający:
  - powiązania stan-widok (view-state)
  - przejścia między stanami (transitions)
  - definicje stanu startowego i stanów końcowych
  - definicje akcji

JAVATECH

Stan

- Każdy stan ma swoje id
- Do definiowania stanów używa się elementu view-state:

```
<view-state id="enterTransferDetails"/>
```

- Pierwszy zdefiniowany stan jest domyślnie stanem startowym
- Stan końcowy wyróżnia się elementem end-state:

```
<end-state id="transferAccepted"/>
<end-state id="transferCancelled"/>
```

- Po dotarciu do któregoś stanu końcowego, przepływ się kończy i zwraca rezultat

Powiązanie stanu z widokiem

- Atrybut "view"
  - Domyślnie przyjmuje lokalizację szablonu widoku względem lokalizacji pliku z definicją przepływu
  - W przypadku użycia MvcViewFactoryCreatora, przyjmuje logiczną nazwę widoku Spring MVC

```
<view-state id="enterTransferDetails"
        view="transferDetails" />
```

Przejścia pomiędzy stanami

- Przejścia są uzależnione od zdarzenia, które nastąpiło w danym stanie
- Zdarzeniem może być np. wysłanie formularza albo kliknięcie w link

```
<view-state id="enterTransferDetails" view="transferDetails">
    <transition on="submit" to="reviewTransfer"/>
</view-state>

<view-state id="reviewTransfer"  view="reviewTransfer">
        <transition on="confirm" to="transferAccepted" />
        <transition on="revise" to="enterTransferDetails" />
        <transition on="cancel" to="transferCancelled" />
</view-state>

<end-state id="transferAccepted" view="transferAccepted" />
<end-state id="transferCancelled" view="transferCancelled" />
```

Przejścia globalne

- Jedno zdarzenie kieruje do tego samego stanu, niezależnie od stanu źródłowego:

```
<global-transitions>
        <transition on="login" to="login" />
        <transition on="logout" to="logout" />
</global-transitions>
```

JAVATECH

Akcje

- W różnych momentach przepływu, mogą zostać podjęte dodatkowe akcje:
  - on-start: przy rozpoczęciu przepływu
  - on-end: przy zakończeniu przepływu
  - on-entry: przy wejściu do stanu
  - on-exit: przy wyjściu ze stanu
  - on-render: przed zarenderowaniem widoku związanego ze stanem

Wyrażenia i wywoływanie kodu Java

- Tag evaluate służy do obliczenia wartości wyrażenia (albo wywołaniu kodu Java) za pomocą Spring Expression Language:

Wyrażenia i wywoływanie kodu Java

- Tag evaluate służy do obliczenia wartości wyrażenia (albo wywołaniu kodu Java) za pomocą Spring Expression Language:

```
<on-start>
  <evaluate
      expression="transferService.createTransfer(accountId, currentUser.name)" />
</on-start>
```

  - Wynik wyrażenia, może zostać zapisany w modelu związanym z przepływem (flowScope):

```
<on-start>
  <evaluate
      expression="transferService.createTransfer(accountId, currentUser.name)"
      result="flowScope.transfer" />
</on-start>
```

JAVATECH

Zmienne wejściowe

- Zmienne wejściowe pozwalają przekazać dane do przepływu
- Są automatycznie inicjowane z parametrów URI, mogą być przekazywane z przepływów nadrzędnych
- Najprostszy sposób zdefiniowania zmiennej:

```
<input name="accountId"/ >
```

- Można podać też typ (nastąpi konwersja do wskazanego typu):

```
<input name="accountId" type="long" />
```

- Można wymusić podanie wartości:

```
<input name="accountId" required="true" />
```

- Wartości są automatycznie dostępne w modelu widoku i wyrażeniach EL

Zmienne specjalne dostępne w wyrażeniach

- requestScope – kontener na zmienne żyjące przez czas pojedynczego żądania
- flashScope – kontener na zmienne czyszczony po każdym odświeżeniu widoku
- viewScope – kontener na zmienne żyjące przez czas wyświetlania widoku; muszą być Serializable
- flowScope – zmienne globalne dla całego przepływu
- conversationalScope – zmienne globalne dla głównego przepływu i wszystkich przepływów zagnieżdżonych –
  - przechowywane w sesji HTTP
  - muszą być Serializable
- requestParameters – parametry żądania

Zmienne specjalne dostępne w wyrażeniach

- currentEvent – obiekt zdarzenia, które spowodowało przejście
- currentUser – użytkownik, który przeszedł autentykację
- resourceBundle – dostęp do zasobów, np. komunikatów tekstowych
- flowRequestContext – reprezentacja bieżącego żądania
- flowExecutionContext – obiekt opisujący bieżący stan przepływu
- flowExecutionUrl – dostęp do URI dla bieżącego stanu przepływu

Umieszczanie danych w modelu widoku

- Atrybut model określa obiekt danych formularza
  - Obiekt zostaje przekazany do widoku przed wyświetleniem formularza
  - Po wypełnieniu formularza przez użytkownika i wybraniu odpowiedniej akcji, domyślnie następuje walidacja danych i wypełnienie obiektu danymi z formularza

```
<on-start>
  <evaluate expression="new lab.spring.TransferDetails()"
          result="flowScope.transferDetails" />
</on-start>

<view-state id="enterTransferDetails"
          view="transferDetails"
          model="flowScope.transferDetails">
...
</view-state>
```

Umieszczanie danych w modelu widoku c.d.

- Aby umieścić inne obiekty w zasięgu widoku, należy zapisać je wewnątrz viewScope, przed wyświetleniem widoku (akcja on-render):

```
<view-state id="enterTransferDetails"
        view="editTransferDetails"
        model="flowScope.transferDetails">

  <on-render>
        <evaluate expression="accountService.accountDetails(accountId)"
          result="viewScope.accountDetails" />
  </on-render>

</view-state>
```

Kontrola wiązania danych i walidacji

- Czasami chcemy zignorować dane z formularza, np. przy zdarzeniu typu "Anuluj"

```
<transition on="proceed" to="reviewTransfer"/>
<transition on="cancel" to="transferCancelled" bind="false" />
```

  - Do ustawienia odczytu/zapisu tylko wybranych pól obiektu służy tag binder:

```
<binder>
  <binding property="recipient" required="true"/>
  <binding property="recipientAccount" required="true" />
  <binding property="amount" required="true" />
  <binding property="date" />
</binder>
```

Walidacja

- 2 metody walidacji:
  - Za pomocą metody validateStan(ValidationContext) dostarczanej przez model
  - Za pomocą obiektu walidatora (jako bean) o nazwie "NazwaModeluValidator" dostarczającego metody do walidacji poszczególnych stanów o nazwach validateNazwaStanu
- Nie ma jeszcze bezpośredniego wsparcia dla mechanizmów walidacji Spring Framework: Validator i JSR-303

Walidacja – metoda na modelu

```
public class TransferDetails {
    private Date transferDate;
    ...

    public void validateEnterTransferDetails(ValidationContext context) {
        MessageContext messages = context.getMessageContext();
        if (transferDate.before(today())) {
            messages.addMessage(new MessageBuilder().error().source("transferDate").
                defaultText("Transfer date must not be in the past").build());
        }
    }
}
```

Walidacja – osobny walidator

```
public class TransferDetailsValidator {

    public void validateEnterTransferDetails(
            TransferDetails transfer, ValidationContext context) {

        MessageContext messages = context.getMessageContext();
        if (transfer.getTransferDate().before(today())) {
            messages.addMessage(new
            MessageBuilder().error().source("transferDate").
                defaultText("Transfer date must not be in the past").build());
        }
    }
}
```

Generowanie zdarzeń

- Przy zatwierdzaniu formularza – nazwać przesyłany parametr _eventId_NazwaZdarzenia:

```
<input type="submit" name="_eventId_proceed" value="Proceed"/>
<input type="submit" name="_eventId_cancel" value="Cancel"/>
```

- Przez pole ukryte:

```
<input type="submit" value="Proceed" />
<input type="hidden" name="_eventId" value="proceed" />
```

- Przez kliknięcie w link:

```
<a href="${flowExecutionUrl}&_eventId=cancel">Cancel</a>
```

JAVATECH

Przekazywanie wyników

- Przepływ może zakończyć się w jednym ze stanów końcowych
- Jeśli przepływ był zagnieżdżony, osiągnięcie stanu końcowego oznacza przekazanie sterowania do przepływu rodzica
- Stany końcowe mogą przekazywać wynik wykonania przepływu
- Przepływ rodzic ma dostęp do informacji, w którym stanie końcowym zatrzymał się przepływ zagnieżdżony oraz jakie zwrócił wyniki

Przepływy zagnieżdżone

- Przepływ rodzic – wywołanie przepływu zagnieżdżonego:

```
<subflow-state id="selectRecipient" subflow="selectRecipient">
        <transition on="selected" to="enterTransferDetails">
                <evaluate expression=
         "flowScope.transferDetails.recipient=currentEvent.attributes.recipient"/>
        </transition>
        <transition on="cancelled" to="enterTransferDetails"/>
</subflow-state>
```

- Przepływ zagnieżdżony:

```
<flow ...>
  <end-state id="selected">
        <output name="recipient" value="flowScope.recipient"/>
  </end-state>
  <end-state id="cancelled"/>
</flow>
```

Stany-akcje (action-state)

- Nie są powiązane z widokiem
- Obliczają wartość wyrażenia
- Na podstawie wyniku decydują o kolejnym stanie

```
<action-state id="moreAnswersNeeded">
   <evaluate expression="interview.moreAnswersNeeded()" />
   <transition on="yes" to="answerQuestions" />
   <transition on="no" to="finish" />
</action-state>
```

Stany typu if-then-else (decision-state)

- Użyteczne gdy wymagane tylko rozdzielenie na 2 kolejne stany

```
<decision-state id="moreAnswersNeeded">
   <if
      test="interview.moreAnswersNeeded()"
      then="answerQuestions"
      else="finish" />
</decision-state>
```

JAVATECH

Zarządzanie transakcjami przez przepływ

- Przy uruchamianiu przepływu, tworzona jest nowa sesja Hibernate lub PersistenceContext JPA
- W trakcie działania przepływu obiekty skojarzone z sesją mogą być wielokrotnie edytowane
- Przepływ zarządza wiązaniem tych obiektów z sesją
- Obiekty związane z sesją są zapisywane w bazie gdy przepływ się zakończy
  - <end-state> wymaga wtedy podania atrybutu commit="true"
- Zalecane jest używanie blokowanie optymistycznego

Trwałość przepływów – konfiguracja

- Do definicji przepływu dodać <persistence-context/>
- Do kontekstu Springa dodać JpaFlowExecutionListener albo HibernateFlowExecutionListener:

```
<webflow:flow-executor id="flowExecutor" flow-registry="flowRegistry">
   <webflow:flow-execution-listeners>
      <webflow:listener ref="hbFlowExecutionListener" />
   </webflow:flow-execution-listeners>
</webflow:flow-executor>

<bean id="hbFlowExecutionListener"
     class="org.springframework.webflow.persistence.HibernateFlowExecutionListener">
   <constructor-arg ref="sessionFactory" />
   <constructor-arg ref="transactionManager" />
</bean>
```

Dziedziczenie przepływów

- Dziedziczenie pozwala na wielokrotne użycie tych samych definicji przepływów
- Możliwe na poziomie pojedynczych stanów i całych przepływów
- Przepływ może być oznaczony abstract="true", co powoduje, że nie można tego przepływu używać bezpośrednio, a jedynie za pomocą dziedziczenia
- Możliwe dziedziczenie po kilku przepływach
- Nie można przedefiniowywać odziedziczonych elementów
- Podobne elementy, tj. elementy tego samego typu o tym samym identyfikatorze są łączone (merge)
- Elementy różne są dodawane na końcu, z wyjątkiem: evaluate, persistence-context, bean-import i kilku innych

JAVATECH

Dziedziczenie – przykłady

- Dziedziczenie dwóch przepływów:
  - <flow parent="common-transitions, common-states"> ... </flow>
- Dziedziczenie pojedynczego stanu:
  - <view-state id="child-state" parent="parent-flow#parent-view-state">
- Przepływ abstrakcyjny:
  - <flow abstract="true"> ... </flow>

JAVATECH

Spring Security

JAVATECH

Historia

- projekt powstał w 2003 roku
- dawna nazwa "The Acegi Security System for Spring”
- od 2007 roku jako "Spring Security"

JAVATECH

Podstawowe elementy

- Security Interceptor (zapewnia całościowy mechanizm bezpieczeństwa)
- Authentication manager (identyfikuje użytkownika)
- Access decission manager (rozpoznaje uprawnienia do określonego zasobu)
- Run-as manager (zastępuje tożsamość użytkownika)
- After-invocation manager (kontroluje bezpieczeństwo po uzyskaniu dostępów do zasobów)

AuthenticationProvider

- AuthByAdapterProvider
- AnonymousAuthenticationProvider
- CasAuthenticationProvider
- DaoAuthenticationProvider
- LdapAuthenticationProvider
- RememberMeAuthenticationProvider
- RemoteAuthenticationProvider
- TestingAuthenticationProvider
- X509AuthenticationProvider
- RunAsImplAuthenticationProvider

Konfiguracja

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:security="http://www.springframework.org/schema/security"
  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
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security-3.0.xsd">
    ...
</beans>
```

web.xml

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Domyślna konfiguracja

```
<security:http>
    <security:intercept-url pattern="/**" access="ROLE_USER" />
    <security:form-login />
    <security:anonymous />
    <security:http-basic />
    <security:logout />
    <security:remember-me />
  </security:http>
```

Minimalna konfiguracja

```
<http auto-config='true'>
  <intercept-url pattern="/**" access="ROLE_USER" />
 </http>
```

authentication manager

```
<authentication-manager>
  <authentication-provider>
    <user-service>
      <user name="jimi" password="jimispassword"
                        authorities="ROLE_USER, ROLE_ADMIN" />
      <user name="bob" password="bobspassword"
                        authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
 </authentication-manager>
```

form-login

```
<http auto-config='true'>
  <intercept-url pattern="/login.jsp*" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
  <intercept-url pattern="/**" access="ROLE_USER" />
  <form-login login-page='/login.jsp'/>
 </http>
```

```
<http auto-config='true'>
  <intercept-url pattern="/css/**" filters="none"/>
  <intercept-url pattern="/login.jsp*" filters="none"/>
  <intercept-url pattern="/**" access="ROLE_USER" />
  <form-login login-page='/login.jsp'/>
 </http>
```

Definicja domyślnej strony

```
<http>
  <intercept-url pattern='/login.htm*' filters='none'/>
  <intercept-url pattern='/**' access='ROLE_USER' />
  <form-login login-page='/login.htm' default-target-url='/home.htm'
                                always-use-default-target='true' />
</http>
```

Użycie authentication manager

```
<authentication-manager>
  <authentication-provider user-service-ref='myUserDetailsService'/>
</authentication-manager>

<bean id="myUserDetailsService" class="…"/>
```

Authentication provider to klasa implementująca *UserDetailsService*.

Użycie bazy danych

```
<authentication-manager>
  <authentication-provider>
    <jdbc-user-service data-source-ref="securityDataSource"/>
  </authentication-provider>
 </authentication-manager>
```

alternatywnie

```
<authentication-manager>
  <authentication-provider user-service-ref='myUserDetailsService'/>
 </authentication-manager>


 <beans:bean id="myUserDetailsService"
    class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
  <beans:property name="dataSource" ref="dataSource"/>
 </beans:bean>
```

password encoder

```
<authentication-manager>
  <authentication-provider>
  <password-encoder hash="sha"/>
    <user-service>
      <user name="jimi" password="jimispassword"
                        authorities="ROLE_USER, ROLE_ADMIN" />
      <user name="bob" password="bobspassword"
                        authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
 </authentication-manager>
```

Zarządzanie sesją

```
<http>
  ...
  <session-management invalid-session-url="/sessionTimeout.htm" />
</http>
```

```
  <listener>
   <listener-class>
     org.springframework.security.web.session.HttpSessionEventPublisher
   </listener-class>
  </listener>


<http>
   ...
   <session-management>
       <concurrency-control max-sessions="1" />
   </session-management>
  </http>
```

Zabezpieczanie ziaren

```
<bean id="target"
        class="pl.company.spring.security.service.impl.BankServiceImpl">
        <security:intercept-methods>
                <security:protect method="set*"
                        access="ROLE_ADMIN" />
                <security:protect method="get*"
                        access="ROLE_ADMIN,ROLE_USER" />
        </security:intercept-methods>
</bean>
```