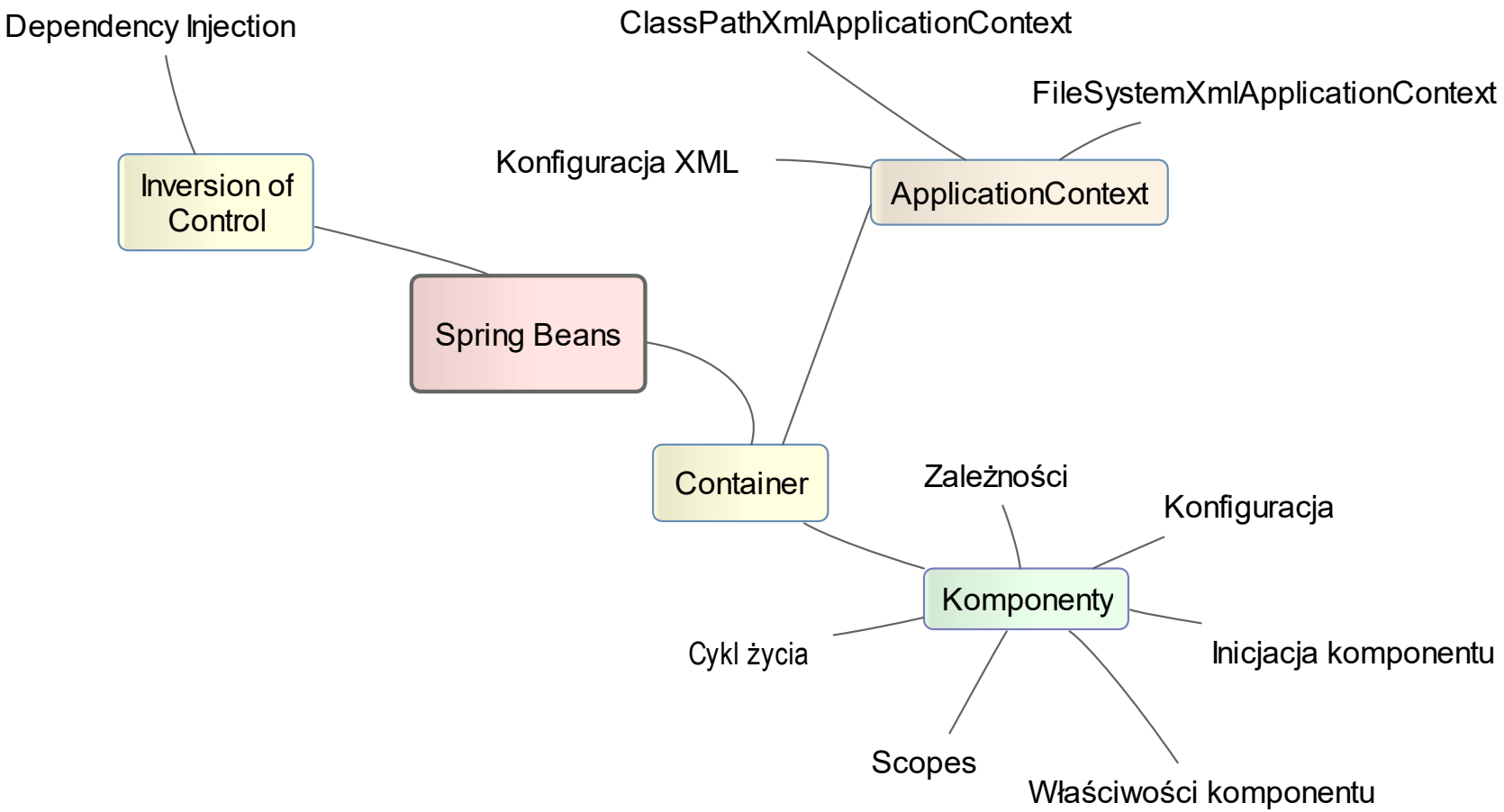
VAVATECH

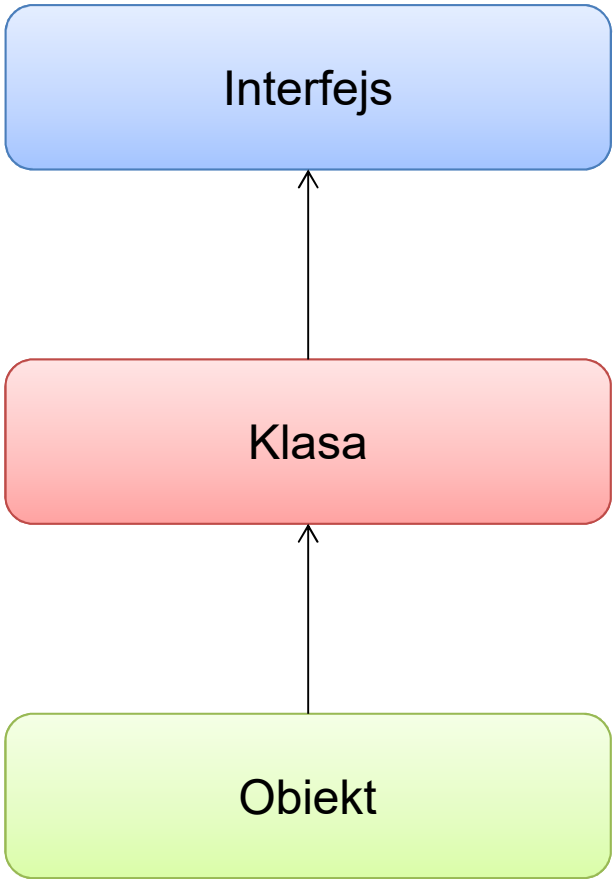
Spring Beans

Realizacja koncepcji Inversion of Control
za pomocą Spring Container

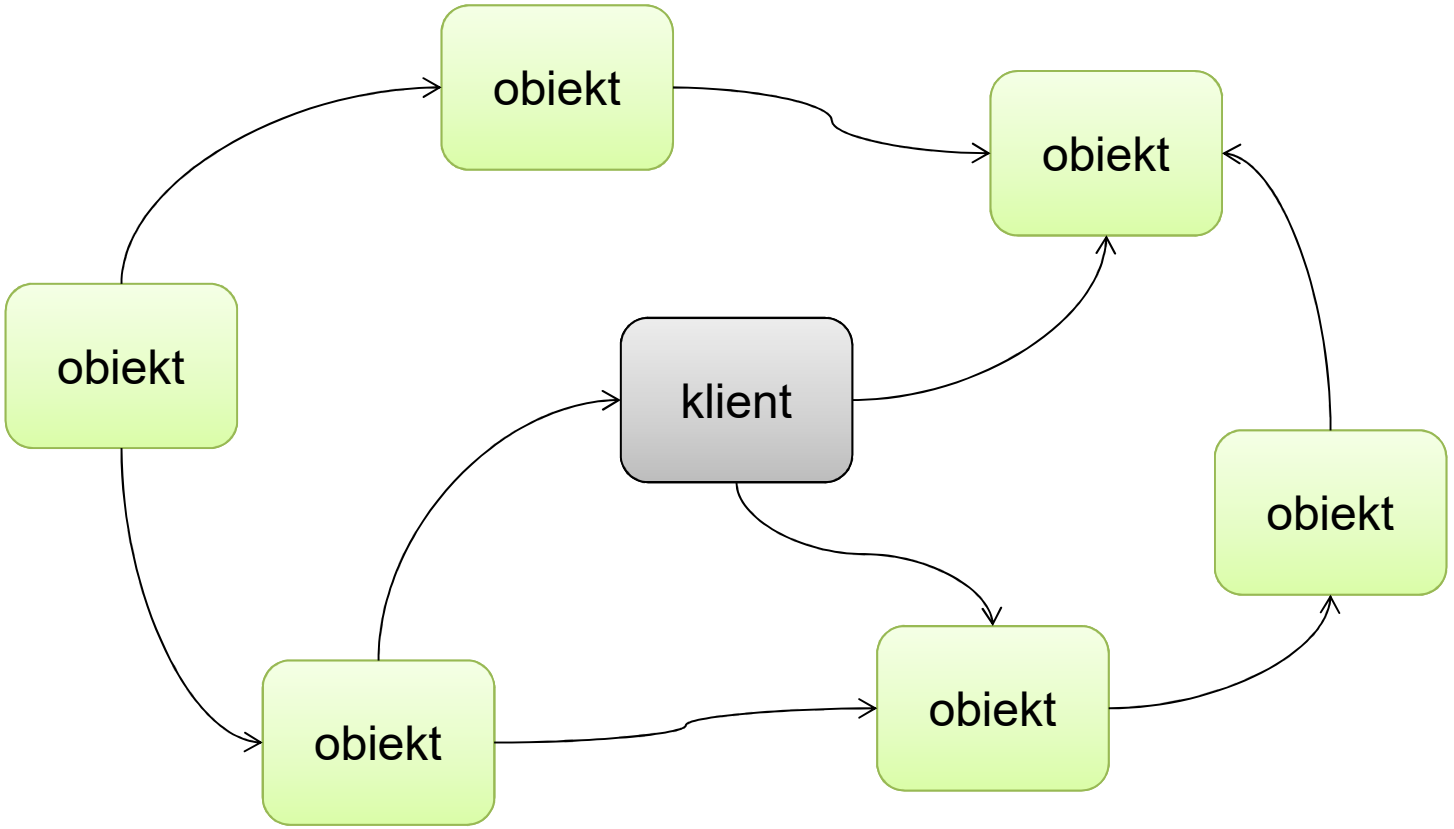
Plan


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

Klasy i obiekty



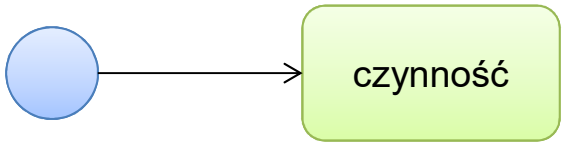
Program komputerowy w Java



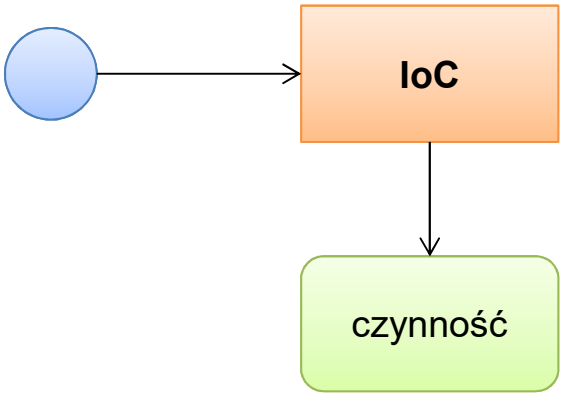
Inversion of Control

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

PODEJŚCIE TRADYCYJNE



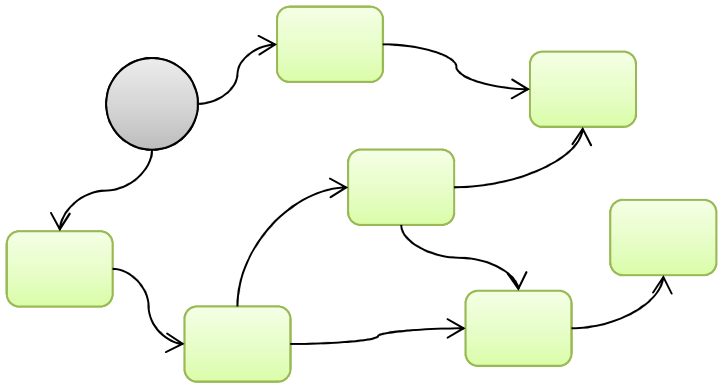
PODEJŚCIE IoC



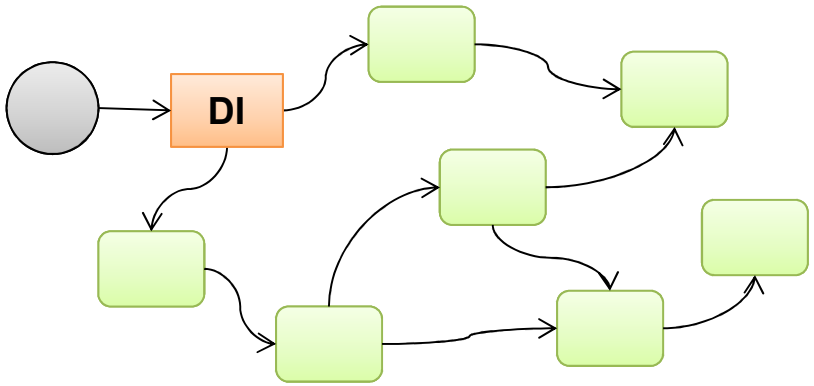
Dependency Injection

- 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



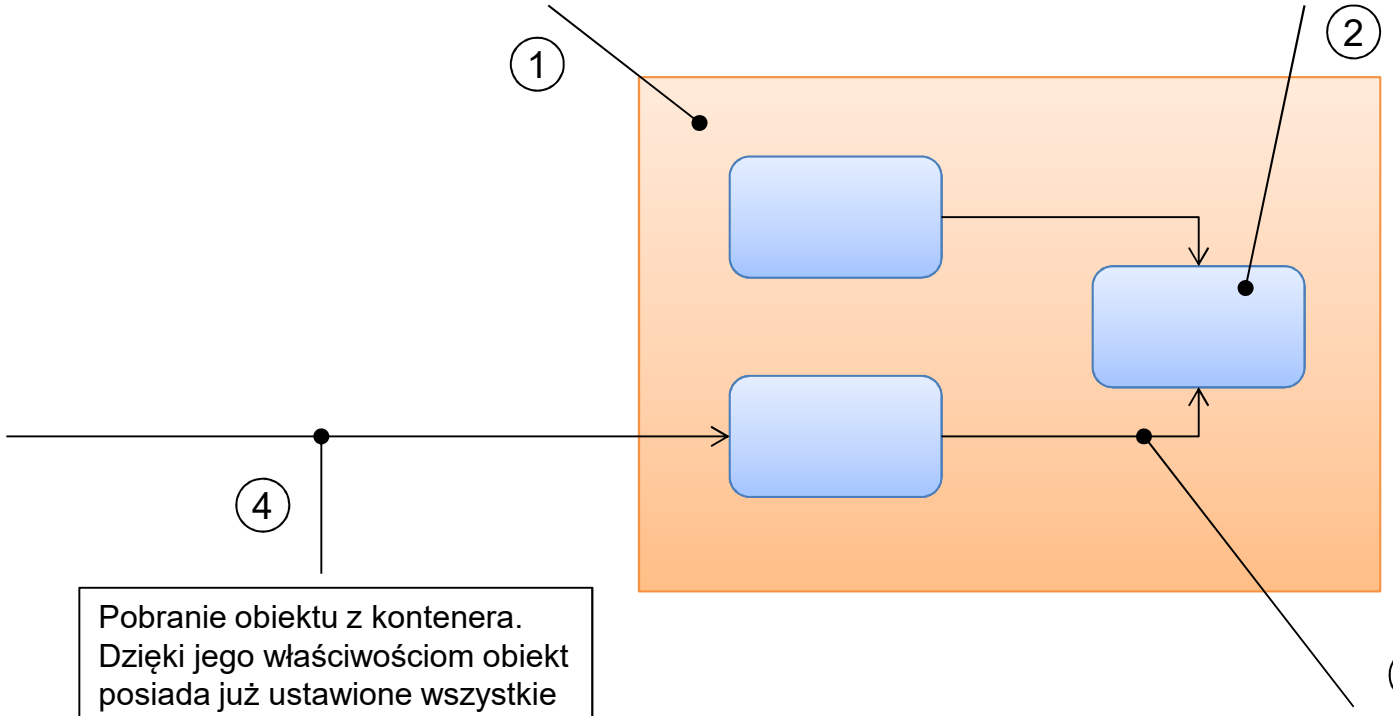
PODEJŚCIE DI



Działanie kontenera DI

Pierwszym krokiem jest powołanie do życia samego kontenera zarządzającego obiektami i zależnościami pomiędzy nimi.

Kontener tworzy instancje obiektów. Klasy oraz sposób powoływania obiektów zdefiniowany jest przez użytkownika.

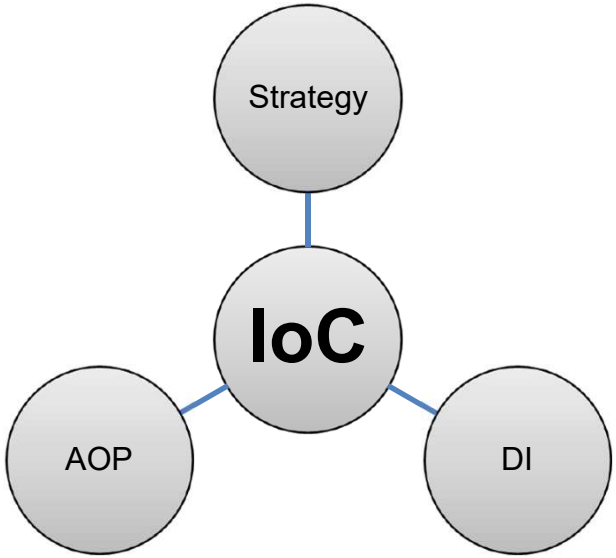


Pobranie obiektu z kontenera. Dzięki jego właściwościom obiekt posiada już ustawione wszystkie zależności do innych obiektów.

Na podstawie warunków zdefiniowanych przez użytkownika kontener ustawia - "wstrzykuje" - zależności pomiędzy komponentami (obiektami).

IoC != DI

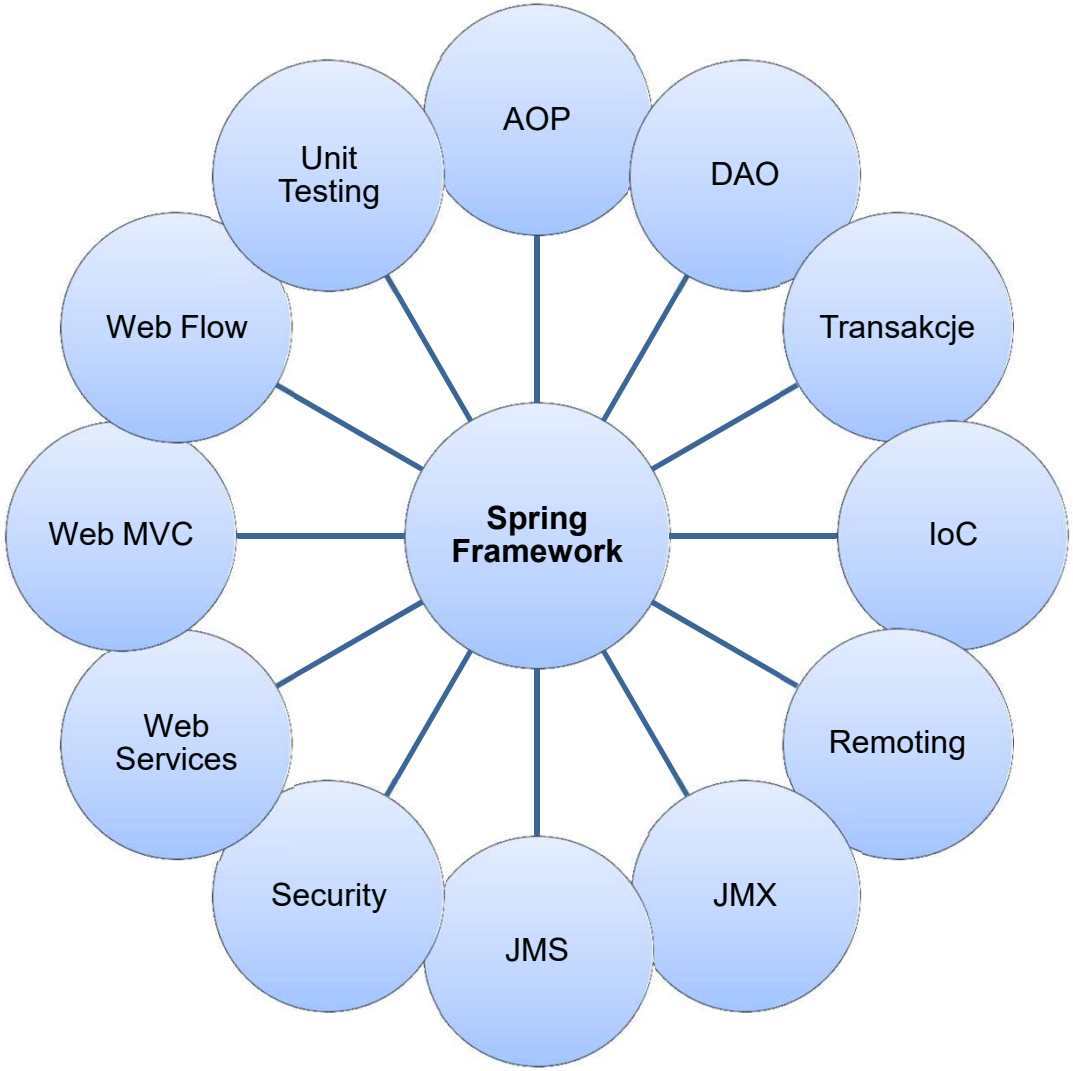
- Inversion of Control często błędnie utożsamiane jest jednoznacznie z Dependency Injection.
- Dependency Injection to szczególny przypadek IoC, który obejmuje szerszy krąg przypadków.



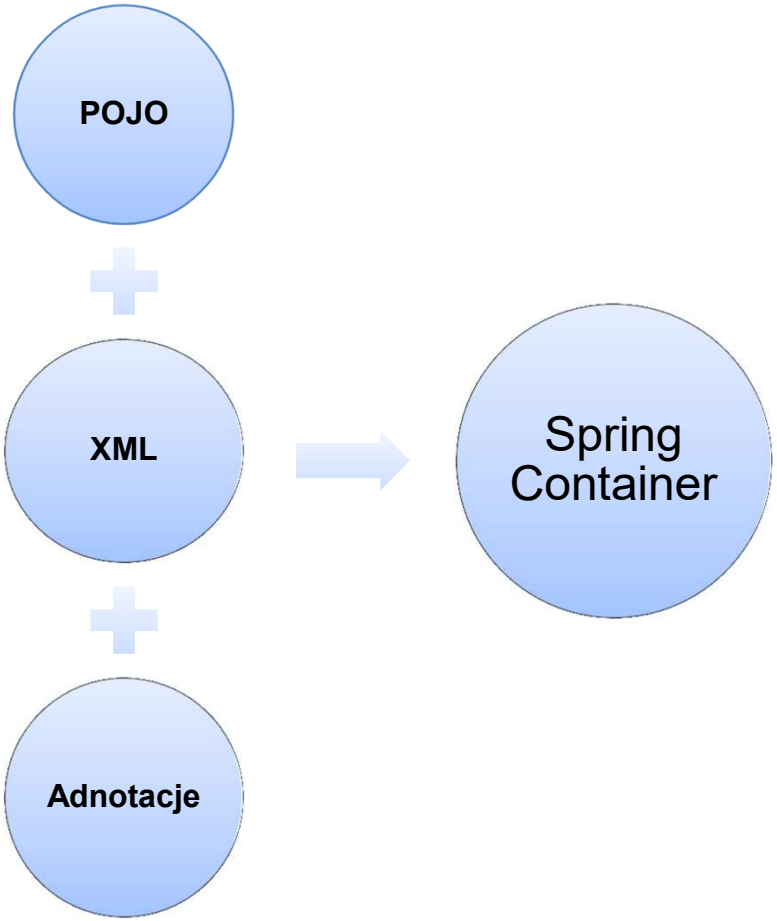
Spring jako alternatywa dla Java EE

- Komponenty EJB (2.1)
  - "ciężkie" i skomplikowane
  - trudne do testowania
- Spring (2003)
  - lekki
  - prosty w konfiguracji
  - łatwy do testowania
  - duża funkcjonalność dodatkowa
- Obecnie
  - aplikacje niezależne od JavaEE (poza Servlet API)
  - łatwa integracja z JavaEE

Spring Framework



Spring jako kontener



XML czy adnotacje?

- Brak jednoznacznej odpowiedzi.
- XML
  - wydaje się być bardziej elastyczny
- Adnotacje
  - silnie typowane potencjalnie pozwalają na uniknięcie błędów
  - konfiguracja za pomocą kodu Java – niektórzy nie lubią XML-a ;)

- Na szczęście możemy używać konfiguracji hybrydowej opartej zarówno o XML jak i adnotacje jednocześnie.

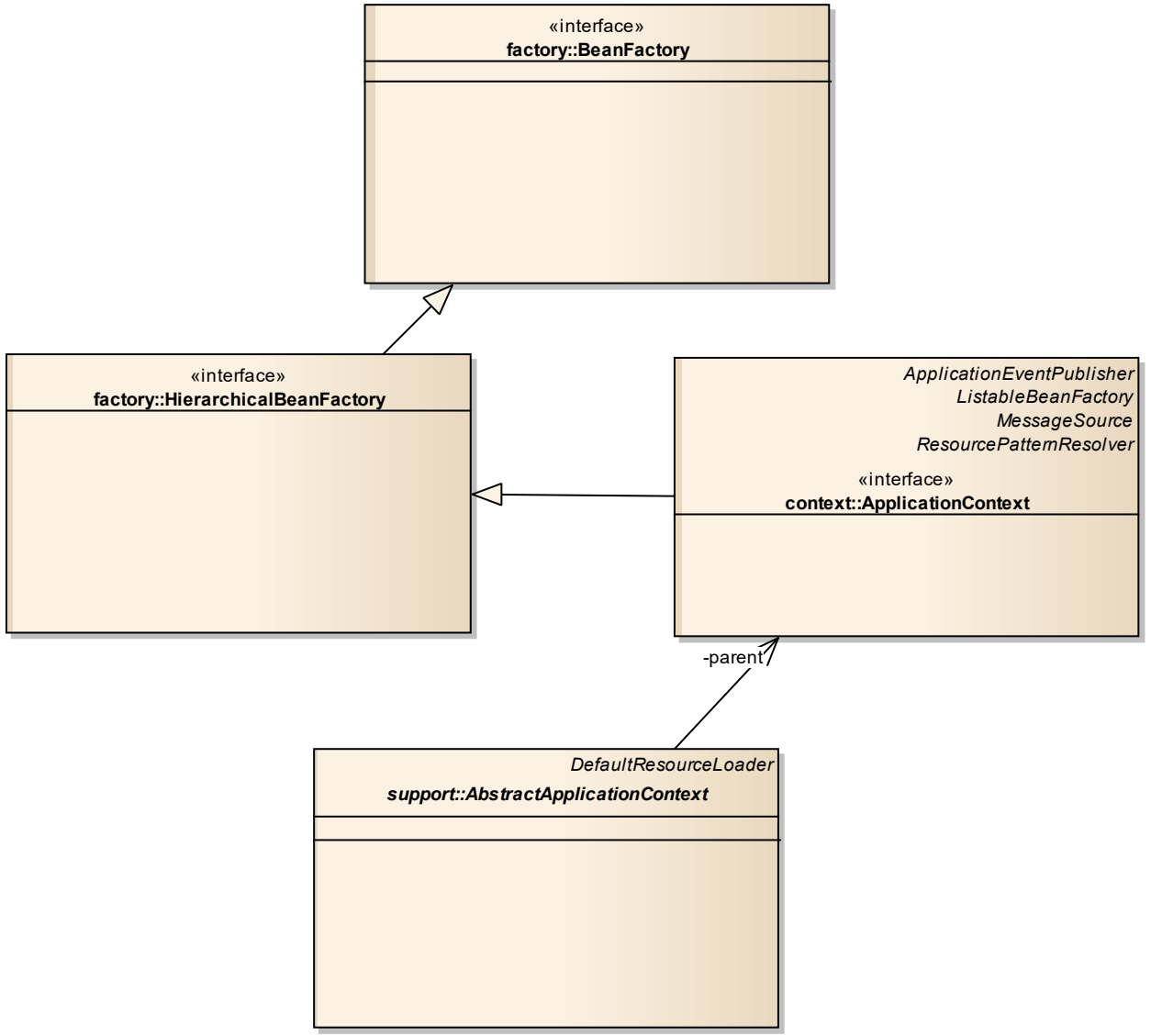
Jak można wykorzystać adnotacje

- Definicja kontenera i komponentów w XML, dodatkowa konfiguracja np. zależności lub cyklu życia w adnotacjach.
- Konfiguracja kontenera w XML, definicja komponentów zarówno w XML jak i za pomocą adnotacji.
- Konfiguracja kontenera za pomocą adnotacji, komponenty zarówno za pomocą adnotacji jak i XML.

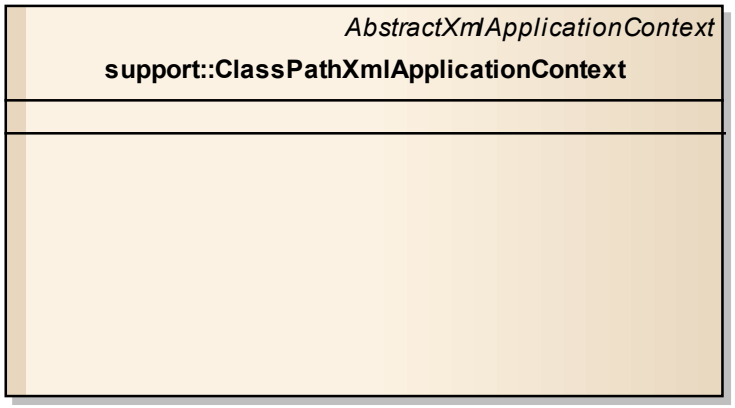
Podstawowe pakiety

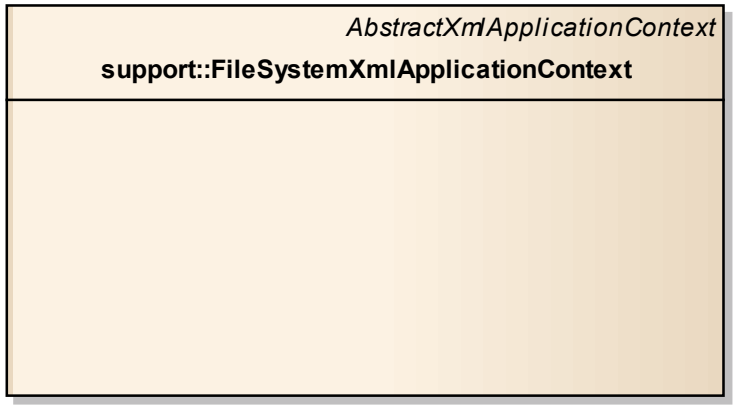
- Konstrukcja springowego IoC osadzona jest w dwóch pakietach
  - org.springframework.beans
  - org.springframework.context

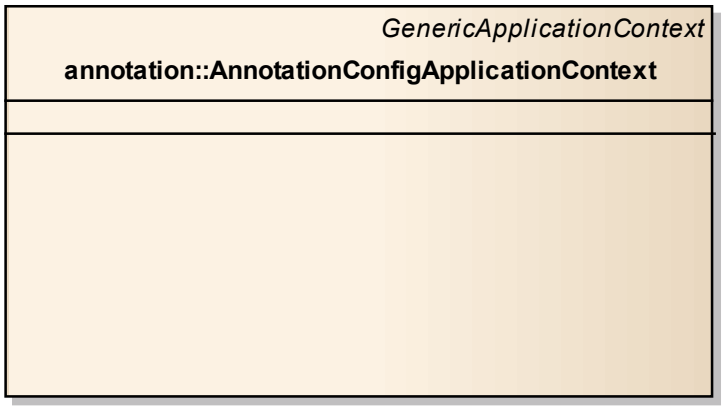
Podstawowe interfejsy



Context Factories

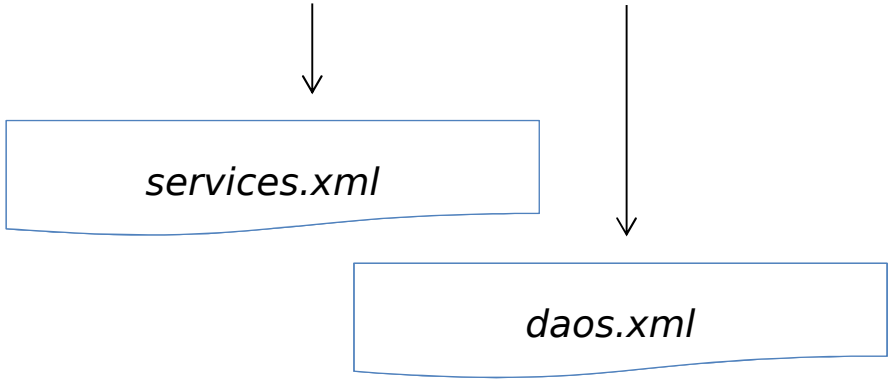






Inicjalizacja kontenera

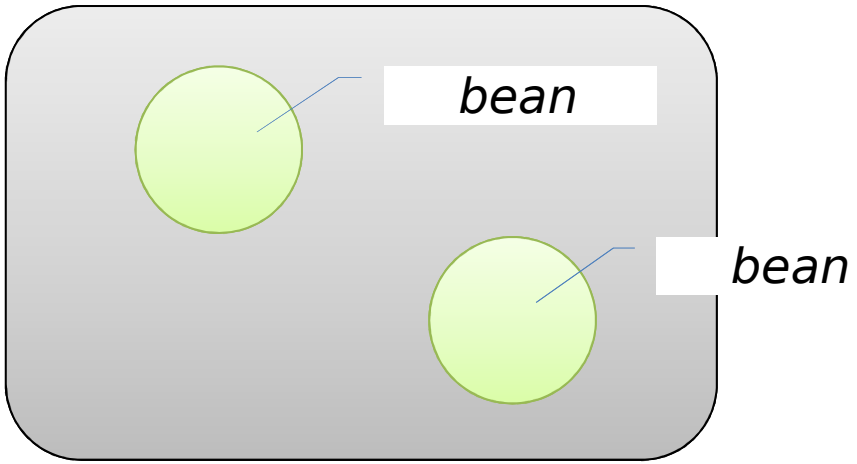
```
ApplicationContext context =
        new ClassPathXmlApplicationContext(
                new String[] {"services.xml", "daos.xml"});
```



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>
```



Rozszerzanie konfiguracji XML

- Rozszerzenie polega na zdefiniowaniu w XML odpowiedniego namespace.
- Namespace skojarzony jest z biblioteką znaczników realizujących określoną funkcjonalność, najczęściej upraszczają proces definiowania komponentów lub ich użycia.
- Istnieje możliwość zdefiniowania własnego namespace i biblioteki znaczników. Może to zdecydowanie uprościć tworzenie i utrzymanie systemów opartych o Spring.

Przykład

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:util="http://www.springframework.org/schema/util"
        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
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util.xsd">

        <!-- <bean/> -->

</beans>
```

Przykładowe namespace

- *xmlns:util*
  - *http://www.springframework.org/schema/util*
- *xmlns:jee*
  - *http://www.springframework.org/schema/jee*
- *xmlns:lang*
  - *http://www.springframework.org/schema/lang*
- *xmlns:tx*
  - *http://www.springframework.org/schema/tx*
- *xmlns:aop*
  - *http://www.springframework.org/schema/aop*
- *xmlns:context*
  - *http://www.springframework.org/schema/context*
- *xmlns:tool*
  - *http://www.springframework.org/schema/tool*

Namespace util

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:util="http://www.springframework.org/schema/util"
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
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd">

        <!-- <bean/> definitions here -->

</beans>
```

Tradycyjne budowanie obiektu listy

```
<bean id="emails"
        class="org.springframework.beans.factory.config
                                        .ListFactoryBean">
    <property name="sourceList">
        <list>
            <value>...</value>
            <value>...</value>
            <value>...</value>
            <value>...</value>
        </list>
    </property>
</bean>
```

Wykorzystanie util:list

```
<util:list id="emails" list-class="java.util.ArrayList">
    <value>...</value>
    <value>...</value>
    <value>...</value>
    <value>...</value>
</util:list>
```

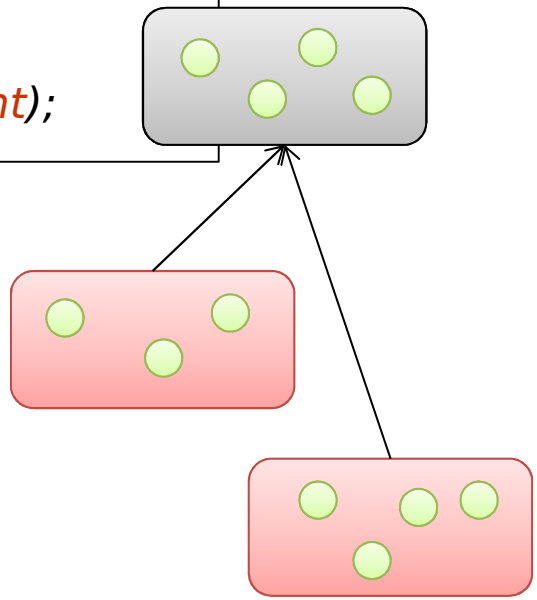
Złożony plik konfiguracyjny

```
<?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

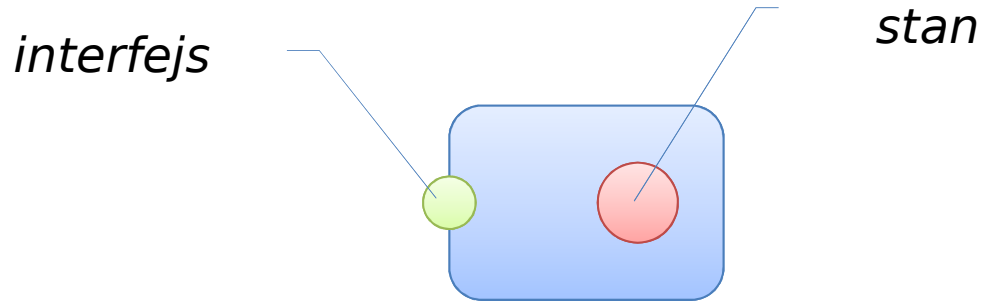
```
ApplicationContext parent =
  new ClassPathXmlApplicationContext(
    new ClassPathResource("/applicationContext.xml"));

ApplicationContext child =
  new ClassPathXmlApplicationContext(
    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



Definicja komponentu

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

Wykorzystanie adnotacji w kontenerze

```
<?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:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="      http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>
    <context:component-scan base-package="com.company"/>

</beans>
```

Definicja komponentu za pomocą adnotacji

- @Component
- @Service
- @Repository
- @Controller

Przykład

```
@Service
public class CompanyServiceImpl
                implements CompanyService {
...
}
```

Identyfikator komponentu

- Nazwa komponentu musi być unikalna w obrębie pojedynczego kontenera, w obrębie którego komponent działa.
- Nazwa nie jest obowiązkowa. Cecha ta wykorzystywana jest w komponentach zagnieżdżonych, technicznych oraz wykorzystując mechanizm autowire.

Identyfikator w XML

- 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.

```
<bean id="companyService"
        name="compService, cService" >
</bean>
```

Identyfikator w adnotacjach

```
@Service("companyService")
public class CompanyServiceImpl
                implements CompanyService {
...
}
```

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 właściwości
- za pomocą statycznej metody fabrykującej
- za pomocą metody fabrykującej obiektu

Bezparametrowy konstruktor

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

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

Właściwości

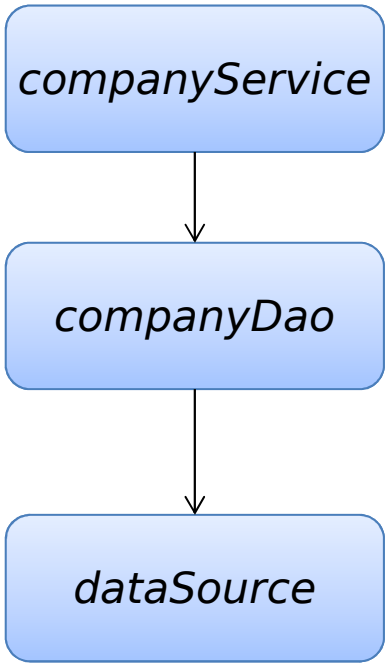
```
public class org.apache.commons.dbcp.BasicDataSource
        implements javax.sql.DataSource {

    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>
```

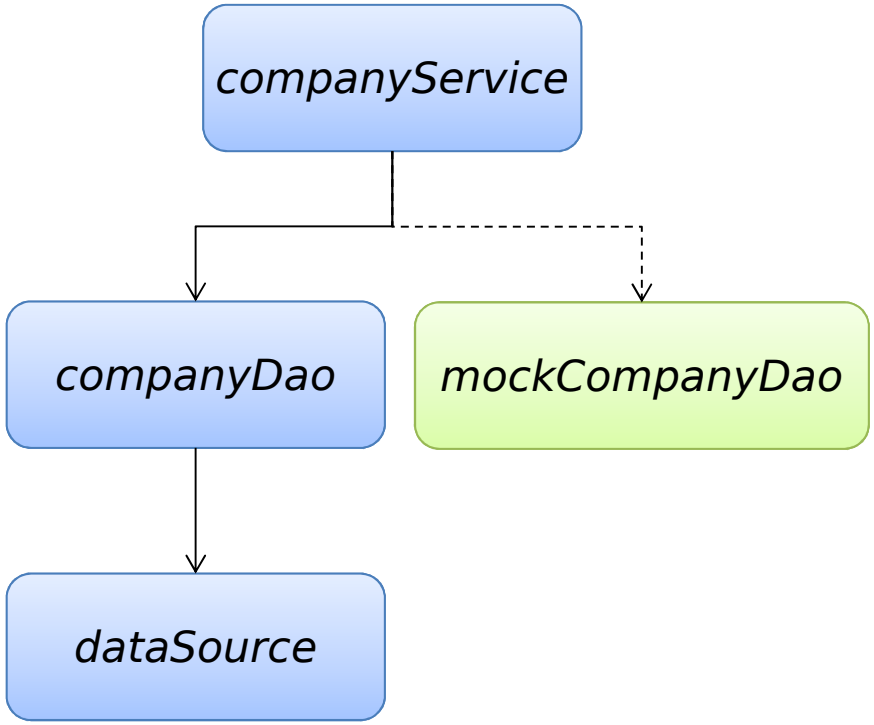
Podanie w konfiguracji właściwości elementu *value* powoduje wywołanie odpowiedniego settera po utworzeniu komponentu w kontenerze. Przykładowo dla własciwości *driverClassName* zostanie wywołana metoda *setDriverClassName.*

Zależności pomiędzy obiektami



System składa się z wielu elementów realizujących różne aspekty logiki biznesowej. Ich kooperacja pozwala na przeprowadzenie operacji dążących do spełnienia wymagań nałożonych na oprogramowanie.
Podział programu na "cegiełki" sprzyja przejrzystości oraz ułatwia utrzymanie.

Interfejsy kontra obiekty



Zależności pomiędzy elementami systemu mogą być realizowane bezpośrednio za pomocą obiektów lub "luźno" przy pomocy interfejsów.

Połączenie przy pomocy interfejsu posiada sporo zalet.
Pozwala na zmianę implementacji określonego interfejsu na inny np. realizujący inaczej daną funkcjonalność, ułatwia testowanie, pozwala na łatwiejsze wprowadzenie AOP.

Referencje

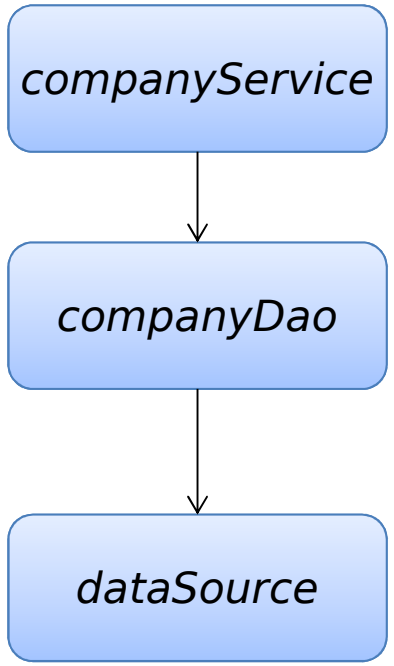
```
public class CompanyServiceImpl
        implements CompanyService {

    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>
```



Namespace p

```
<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.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">


</beans>
```

Przykład

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      p:driverClassName="com.mysql.jdbc.Driver"
      p:url="jdbc:h2:tcp://localhost:3306/mydb"
      p:username="sa" p:password="">

</bean>
```

p-ref

```
<bean id="companyService"
      class="spring.CompanyServiceImpl"
           p:companyDao-ref="companyDao"/>
</bean>


<bean id="companyDao"
      class="dao.CompanyDaoImpl">
           p:dataSource-ref="dataSource"/>
</bean>
```

javax.annotation.Resource

```
@Component
public class CompanyServiceImpl
        implements CompanyService {

@Resource(name="companyDao")
private CompanyDao companyDao;

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

```
@Component
public class CompanyDao Impl
        implements CompanyDao{

        @Resource(name="dataSource")
        private DataSource dataSource;

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

}
```

JSR 330

```
@Named
public class CompanyServiceImpl
        implements CompanyService {

   @Inject
   private CompanyDao companyDao;

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

    }
}
```

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.

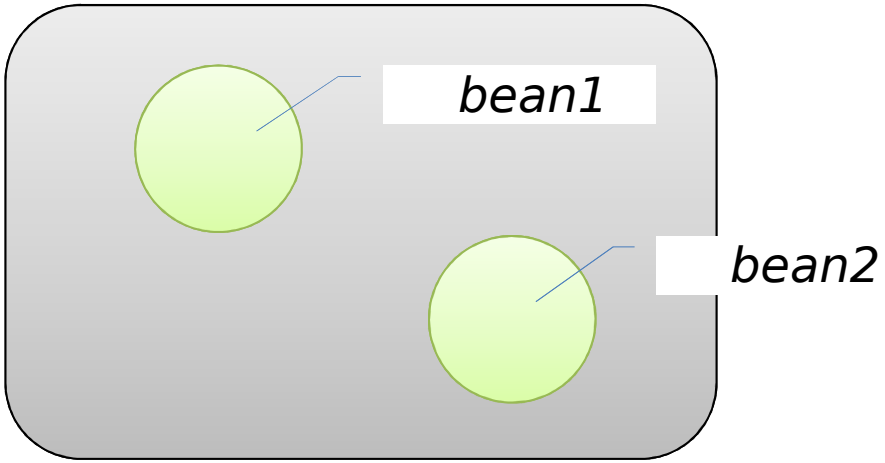
Metoda fabrykująca obiektu

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

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

W tym przypadku nie używany jest atrybut "class".

Pobranie komponentu z kontenera

```
ApplicationContext ctx = ….. ;

MyObject obj = (MyObject) ctx.getBean("bean1");

MyObject obj = ctx.getBean("bean1", MyObject.class);

Map<String, MyObject > map =  ctx.getBeansOfType(MyObject .class);
```

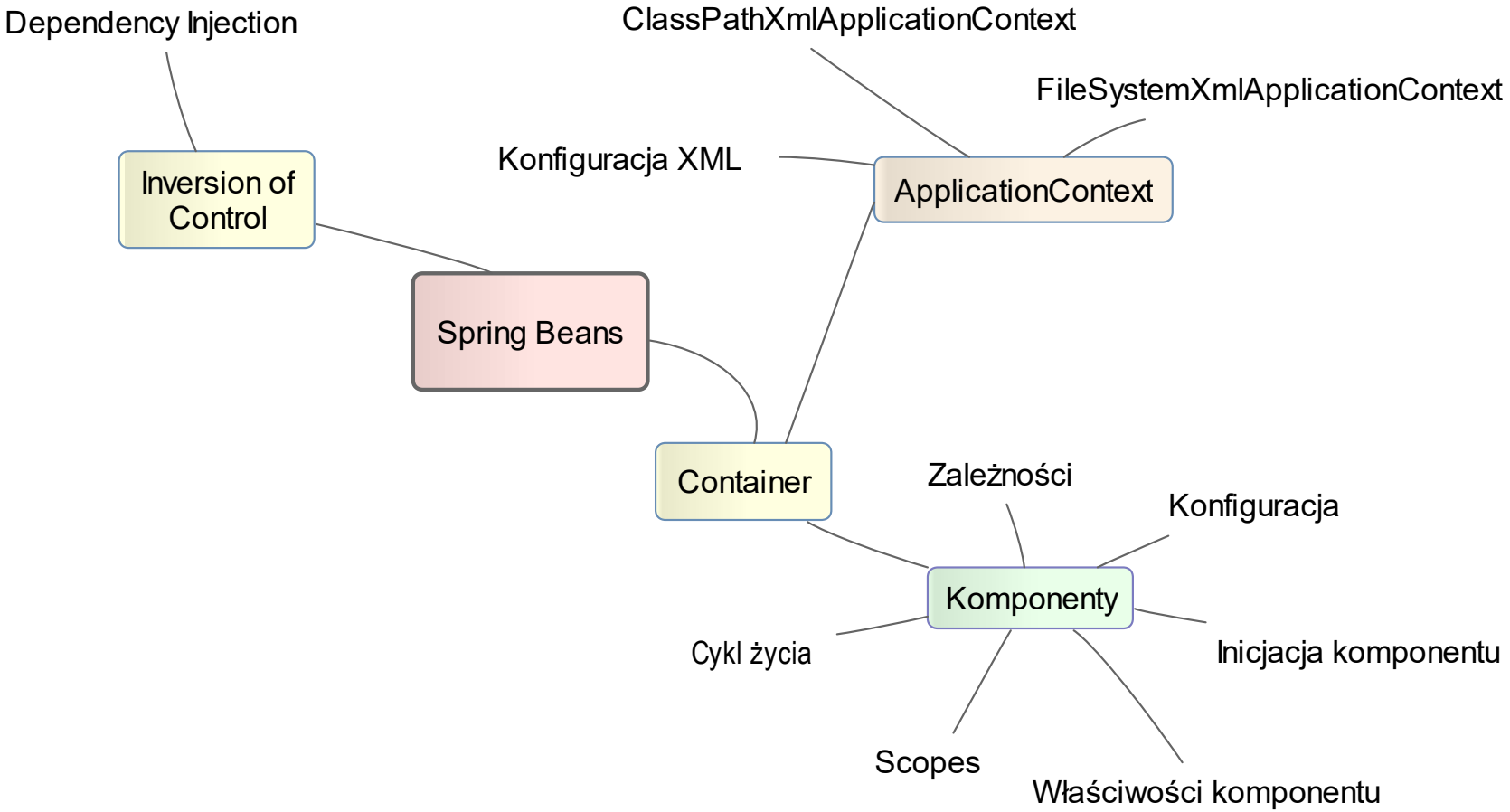


VAVATECH

Spring Beans

Realizacja koncepcji Inversion of Control
za pomocą Spring Container

Plan


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

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

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

```
@Component
public class CompanyServiceImpl
        implements CompanyService {

@Autowired
private CompanyDao companyDao;

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

```
@Component
public class CompanyDao Impl
        implements CompanyDao{

        @Autowired
        private DataSource dataSource;

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

}
```

@Autowired

```
@Component
public class CompanyServiceImpl
        implements CompanyService {

@Autowired
@Qualifier("main")
private CompanyDao companyDao;

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

}
```

```
<bean class="lab.spring.CompanyDaoImpl">
  <qualifier value="main"/>
</bean>
```

@Autowired

```
@Component
public class CompanyServiceImpl
        implements CompanyService {

@Autowired
@Qualifier("main")
private CompanyDao companyDao;

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

```
@Component
@Qualifier("main")
public class CompanyDao Impl
        implements CompanyDao{

        @Autowired
        private DataSource dataSource;

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

Własny @Qualifier

```
@Target({ElementType.FIELD,
        ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface MainDao {



}
```

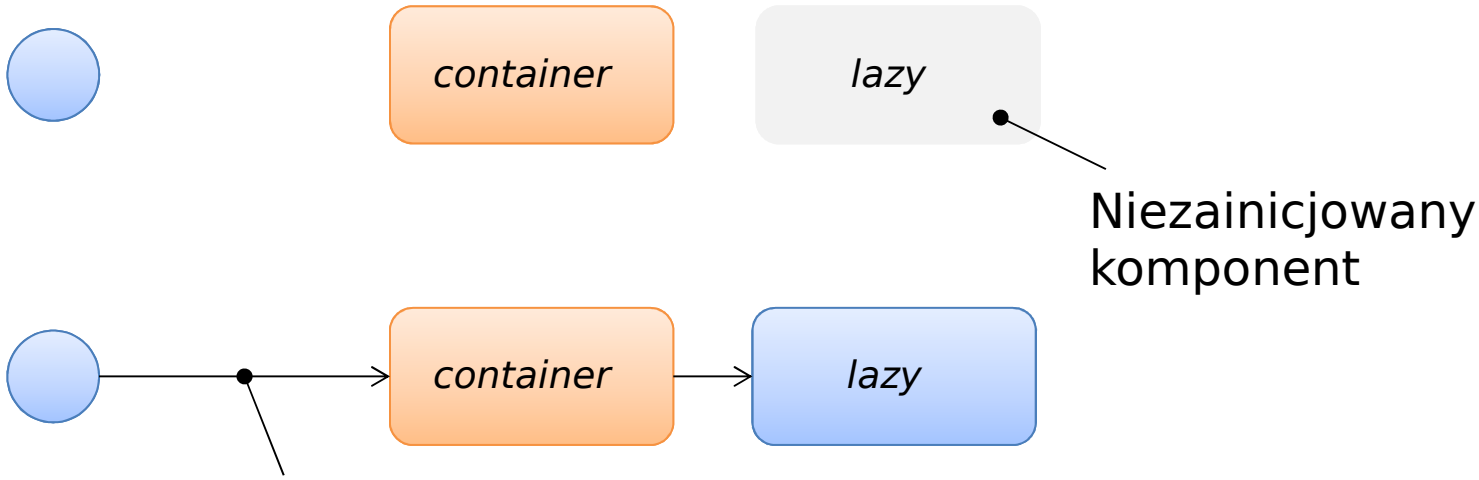
@Autowired

```
@Component
public class CompanyServiceImpl
        implements CompanyService {

@Autowired
@MainDao
private CompanyDao companyDao;

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

```
@Component
@MainDao
public class CompanyDao Impl
        implements CompanyDao{

        @Autowired
        private DataSource dataSource;

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

lazy-init

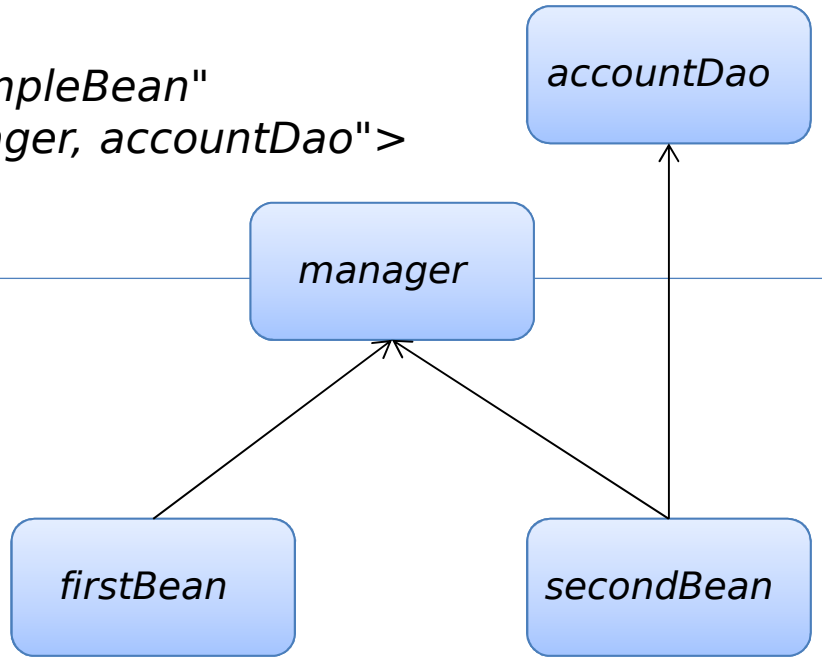
```
<bean id="lazy"
      class="lab.spring.ExpensiveToCreateBean"
      lazy-init="true"/>
```



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>
```



@Lazy @DependsOn

```
@Service("companyService")
@Lazy("true")
@DependsOn({"companyDao"})
public class CompanyServiceImpl
                implements CompanyService {
...
}
```

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>
```

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.

util:map i util:set

```
<util:map id="emails">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>

<util:set id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

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/>

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 {

   }

}
```

Obsługa za pomocą adnotacji

```
@PostConstruct
public void init(){

}

@PreDestroy
public void destroy(){

}
```

Bean scopes

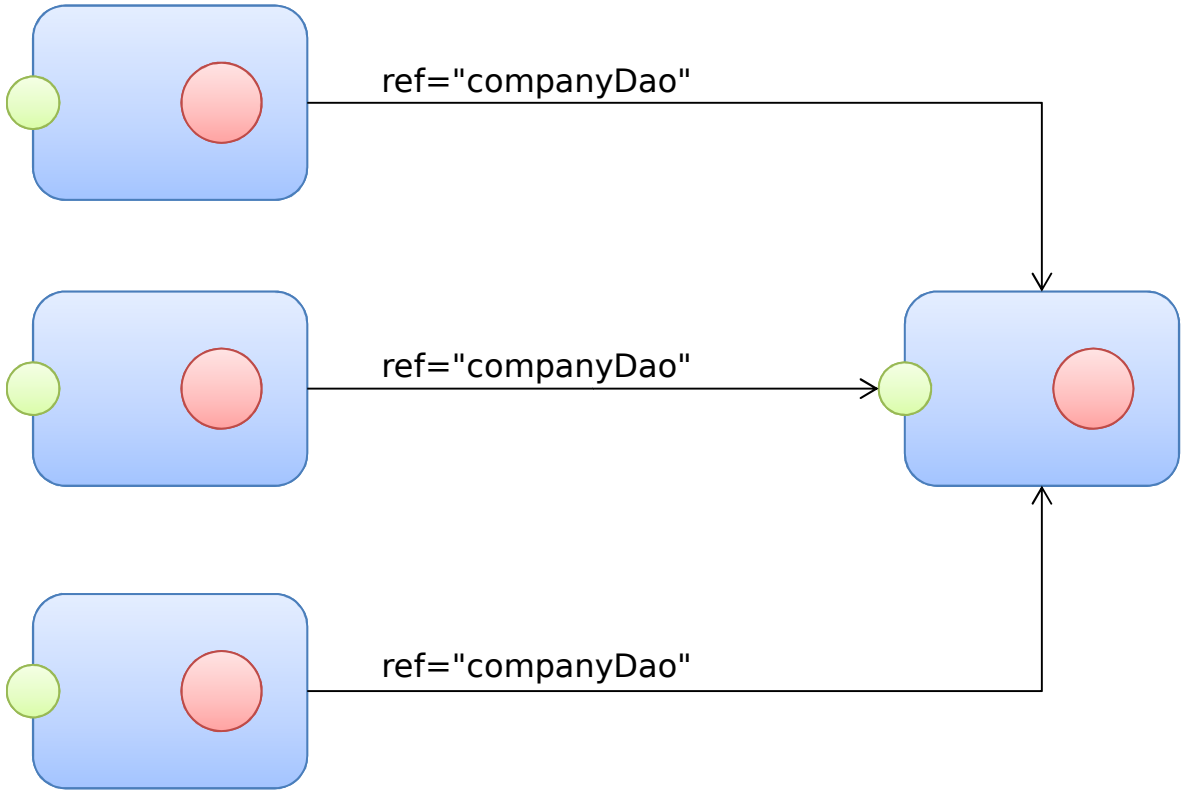
- singleton (domyślnie)
- prototype
- session
- request
- globalScope
- własne

Definicja scope

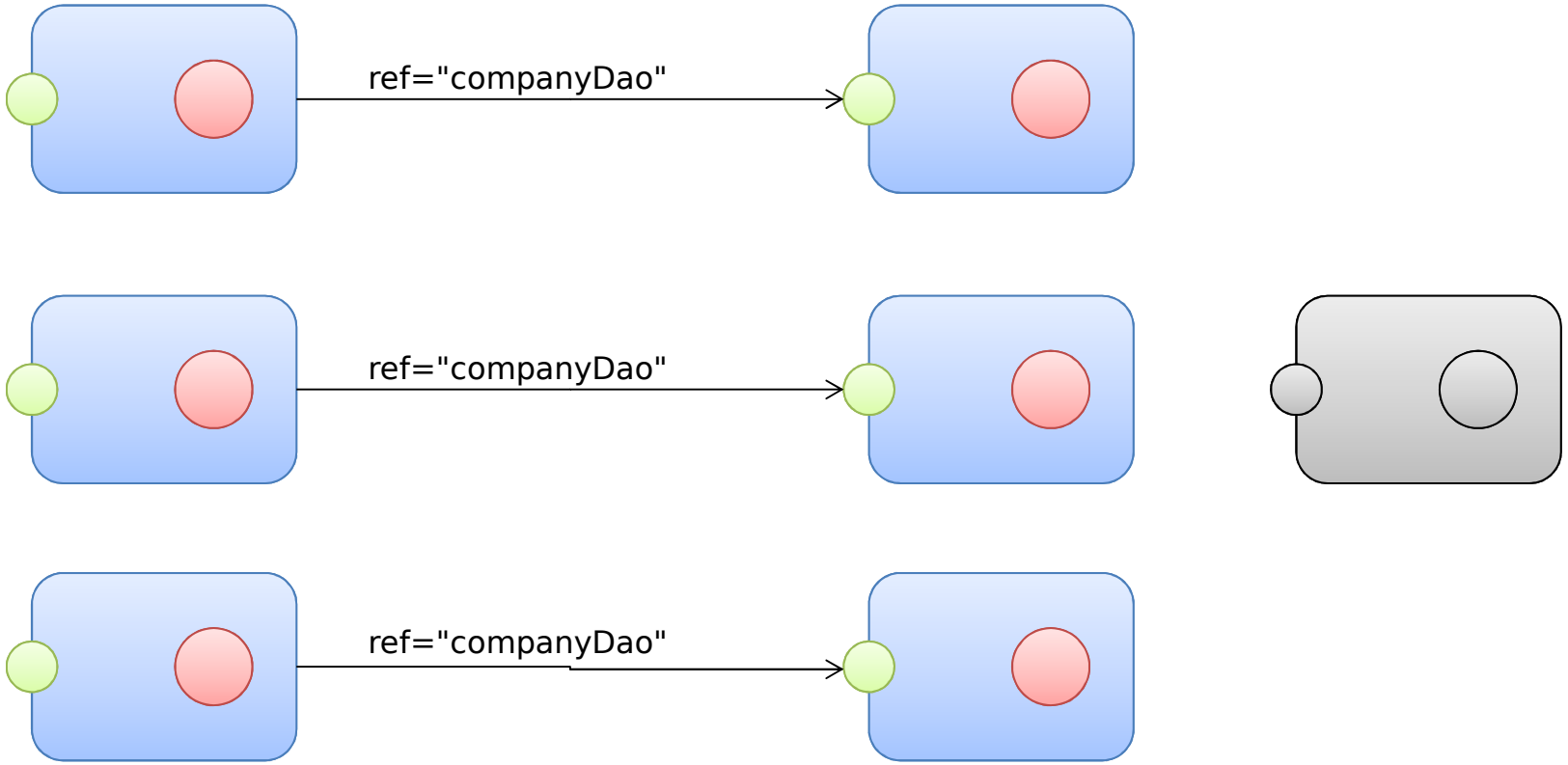
```
<bean id="personService"
        class="lab.spring.CompanyServiceImpl"
                scope="prototype"/>
```

```
@Service("companyService")
@Scope("prototype")
public class CompanyServiceImpl
                implements CompanyService {
...
}
```

Singleton



Prototype



Interfejs Scope

```
package org.springframework.beans.factory.config;

public interface Scope {

    public Object get(String name, ObjectFactory<?> objectFactory);

    public Object remove(String name);

    public void registerDestructionCallback(String name, Runnable callback);

    public Object resolveContextualObject(String key);

    public String getConversationId();

}
```

Rejestracja scope

```
<bean
        class="org.springframework.beans.factory.config
                                        .CustomScopeConfigurer">
    <property name="scopes">
        <map>
            <entry key="myScope">
                <bean class="lab.java.spring.scope.MyScope"/>
            </entry>
        </map>
    </property>
</bean>
```

Użycie

```
<bean id="scopedCompanyService"
      class="lab.java.spring.beans
                    .CompanyServiceImpl"
      scope="myScope">
     <aop:scoped-proxy
           proxy-target-class="false"/>
  </bean>
```

Resource

```
public interface Resource extends InputStreamSource {
    boolean exists();
    boolean isOpen();
    URL getURL() throws IOException;
    File getFile() throws IOException;
    Resource createRelative(String relativePath)
        throws IOException;
    String getFilename();
    String getDescription();
}
```

Standardowe implementacje

- UrlResource
- ClassPathResource
- FileSystemResource
- ServletContextResource
- InputStreamResource
- ByteArrayResource

Pobranie Resource

```
Resource template = ctx.getResource(
        "some/resource/path/myTemplate.txt");

Resource template = ctx.getResource(
        "classpath:some/resource/path/myTemplate.txt");

Resource template = ctx.getResource(
        "file:/some/resource/path/myTemplate.txt");

Resource template = ctx.getResource(
        "http://myhost.com/resource/path/myTemplate.txt");
```

Zasoby jako zależności

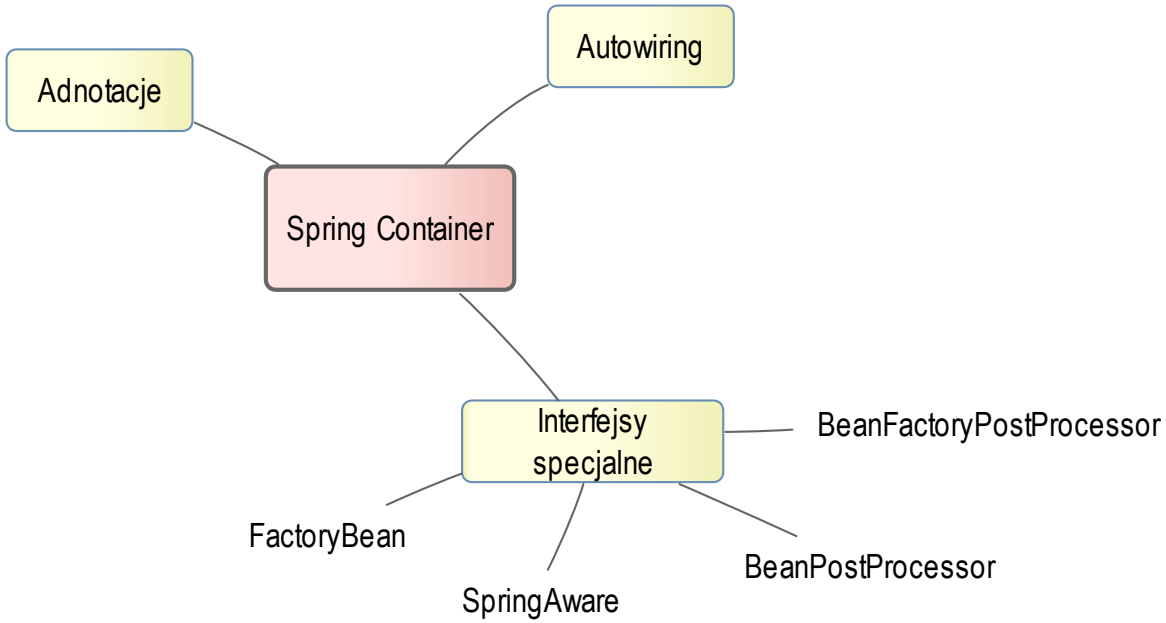
```
<bean id="myBean" class="...">
        <property name="template"
                value="some/resource/path/myTemplate.txt"/>
</bean>

<property name="template"
                value="classpath:
                        some/resource/path/myTemplate.txt">

<property name="template"
                value="file:
                        /some/resource/path/myTemplate.txt"/>
```

VAVATECH

Spring Container

Konfigurowanie kontenera
i komponentów za pomocą kodu Java

Plan


Adnotacje
Autowiring
Spring Container
Interfejsy specjalne
BeanFactoryPostProcessor
FactoryBean
SpringAware
BeanPostProcessor

@Bean

- Adnotacją @Bean można oznaczyć metodę w klasie komponentu.
- Zwracany obiekt zostaje umieszczony w kontenerze Spring.
- W adnotacji można m.in. określić nazwę dla komponentu.

```
@Bean(name = "companyService")
 public CompanyService getCompany() {
        return new CompanyServiceImpl();
 }
```

Bezpośrednie wywoływanie metod

```
@Bean(name = "companyDao")
 public CompanyDao getCompanyDao() {
        return new CompanyDaoImpl();
 }


@Bean(name = "companyService")
 public CompanyService getCompany() {
        return new CompanyServiceImpl(getCompanyDao());
 }
```

UWAGA – może to spowodować komplikacje !!!

@Configuration

- Kontener Spring może zostać zainicjowany za pomocą specjalnej klasy Java.

```
@Configuration
public ApplicationContextConfiguration{

}
```

Przykład

```
@Configuration
public ApplicationContextConfiguration{

    @Bean(name = "companyDao")
     public CompanyDao getCompanyDao() {
            return new CompanyDaoImpl();
     }

    @Bean(name = "companyService")
     public CompanyService getCompany() {
            return new
          CompanyServiceImpl(getCompanyDao());
     }
}
```

Teraz obiekty będą prawidłowo zagnieżdżone

Inicjalizacja kontenera

```
ApplicationContext ctx =
        new AnnotationConfigApplicationContext(
                ApplicationContextConfiguration.class);
CompanyService companyService =
        ctx.getBean(CompanyService.class);
```

Podział konfiguracji na klasy

```
AnnotationConfigApplicationContext ctx =
        new AnnotationConfigApplicationContext();
ctx.register(ApplicationContextConfiguration.class);
ctx.register(AdditionalConfig.class);
ctx.refresh();
CompanyService companyService =
        ctx.getBean(CompanyService.class);
```

Tworzenie komponentów

```
@Configuration
public ApplicationContextConfiguration{

    @Bean
     public CompanyDao companyDao() {
            return new CompanyDaoImpl();
     }

}
```

Gdy w adnotacji @Bean nie określimy nazwy to zostanie przypisana na podstawie nazwy metody – w tym przypadku "*companyDao*"

Dodatkowe opcje

```
@Configuration
public ApplicationContextConfiguration{

    @Bean(
        name={"companyDao", "aliasedCompanyDao"},
        init-method = ”init”,
        destroy-method = ”destroy”,
        autowire=Autowire.NO)
     public CompanyDao companyDao() {
          return new CompanyDaoImpl();
     }

}
```

Autowire.NO
Autowire.BY_NAME
Autowire.BY_TYPE

Dodatkowe adnotacje

```
@Bean(name = "companyDao")
@DependsOn
@Lazy
@Profile({"development"})
@Scope(”prototype”,
        proxyMode=ScopedProxyMode.DEFAULT)
public CompanyDao getCompanyDao() {
    return new CompanyDaoImpl();
}
```

ScopedProxyMode.DEFAULT
ScopedProxyMode.NO
ScopedProxyMode.INTERFACES
ScopedProxyMode.TARGET_CLASS

@Primary

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

Złożona konfiguracja

```
@Configuration
@Import({"AdditionalConfig.class, OtherClass.class})
@ImportResource({"applicationContext.xml"})
public ApplicationContextConfiguration{


}
```

XML centric

```
@Configuration
public ApplicationContextConfiguration{
}
```

```
<beans>
<context:annotation-config/>
<context:property-placeholder
        location="classpath:/lab/spring/jdbc.properties"/>
<bean class="lab.spring. ApplicationContextConfiguration"/>
</beans>
```

```
<beans>
<context:component-scan base-package="lab.spring"/>
<context:property-placeholder
        location="classpath:/lab/spring/jdbc.properties"/>
</beans>
```

Nadawanie wartości własnościom - @Value

```
@Configuration
public class ApplicationContextConfiguration{
    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);
    }
}
```

Environment

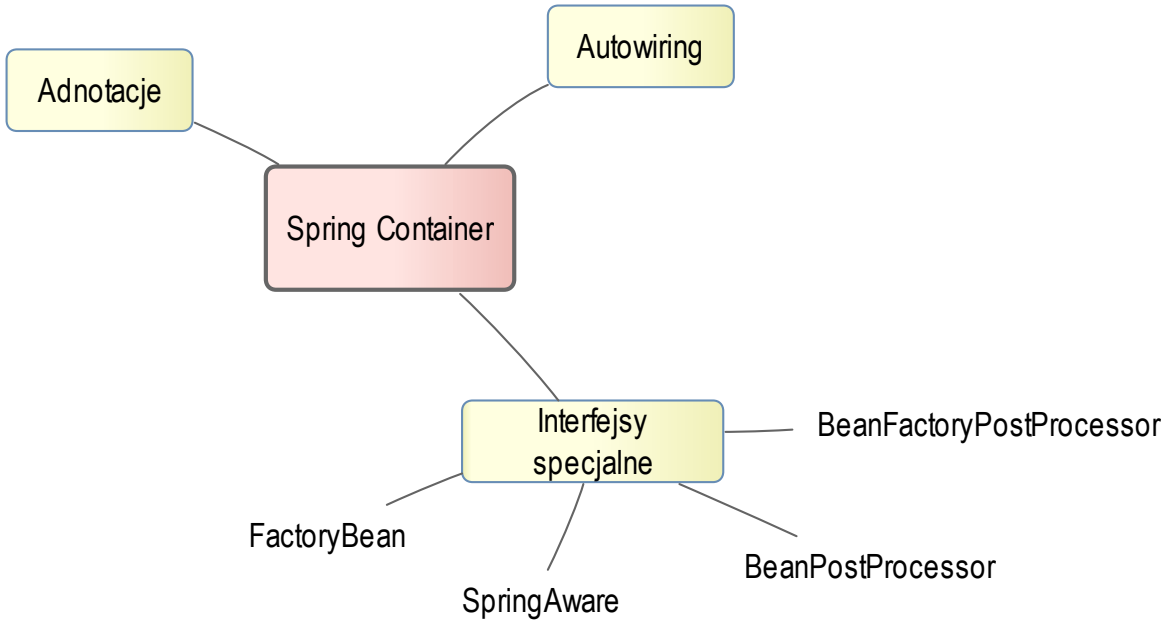
```
@Configuration
@PropertySource("/lab/spring/jdbc.properties")
public class ApplicationContextConfiguration{
    @Autowired
   private Environment env;

    public @Bean DataSource dataSource() {
     return new DriverManagerDataSource(
          env.getProperty("jdbc.url"),   env.getProperty("jdbc.username"),
          env.getProperty("jdbc.password")
          );
    }
}
```

VAVATECH

Spring Container

Zaawansowane funkcje kontenera komponentów
Spring Framework

Plan


Adnotacje
Autowiring
Spring Container
Interfejsy specjalne
BeanFactoryPostProcessor
FactoryBean
SpringAware
BeanPostProcessor

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

VAVATECH

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()

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;
    }
}
```

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

PropertyOverrideConfigurer

```
<bean
  class="org.springframework.beans.factory.config.PropertyOverrideConfigurer">
        <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="user"/>
  <property name="password" value="password"/>
</bean>
```

```
dataSource.username=sa
dataSource.password=root
```

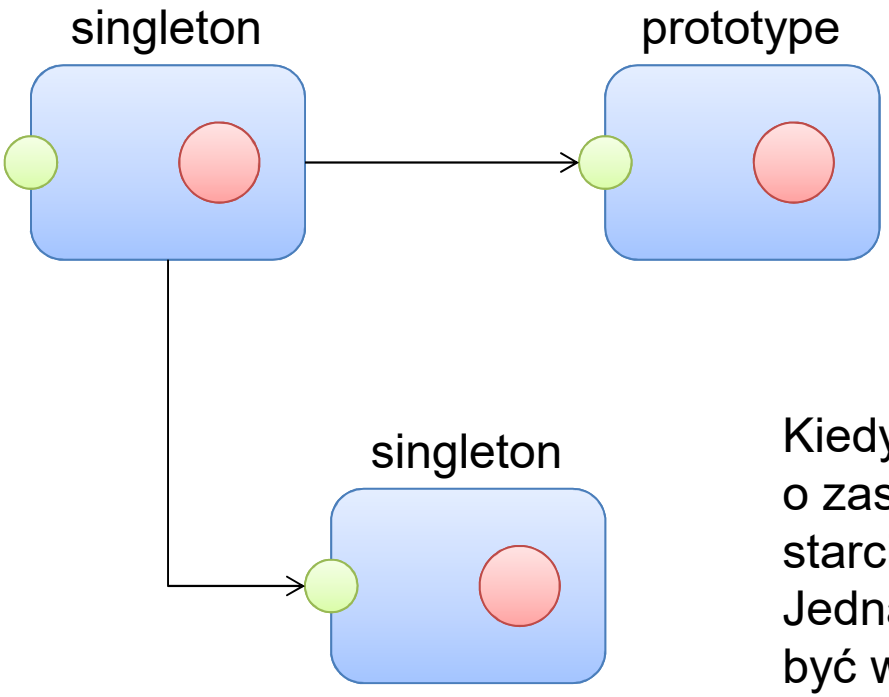
BeanNameAware

```
public class PersonServiceImpl
        implements PersonService, BeanNameAware {

        public void setBeanName(String name) {

        }

}
```

ApplicationContextAware

```
public class PersonServiceImpl
        implements PersonService, ApplicationContextAware {

        public void setApplicationContext(ApplicationContext ctx)
                throws BeansException {

        }

}
```

Zależności – potencjalny problem



Kiedy wstrzykujemy komponent o zasięgu prototype tylko raz na starcie kontenera to nie ma problemu. Jednak komponent prototypowy może być wykorzystany np. jako podstawowy szablon obiektu takiego jak komunikat o zdarzeniu, mail itp..

Rozwiązanie pierwsze

- Można skorzystać z ApplicationContextAware.
- Wymaga to wprowadzenia zależności kodu od elementów Springa.
- Wymaga to podania nazwy komponentu w kodzie i nie jest to już IoC

Rozwiązanie drugie - wstrzykiwanie metod

- Kontener może dostarczyć implementację zadanej metody w naszym kodzie.
- Implementowana metoda będzie zwracała komponent tak jak skonfigurowane w pliku XML
- Metoda musi mieć postać:

  <public|protected> [abstract] <zwracany-typ> nazwaMetody();

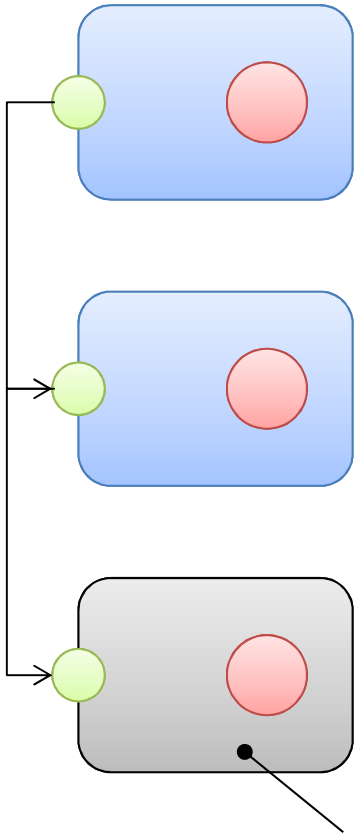
Przykład klasy

```
public abstract class Hello {
   public void printHello {
    TextTemplate textTemplate = createTextTemplate();
    String text = textTemplate.process(Object ... params);
   }
    protected abstract TextTemplate createTextTemplate();
}
```

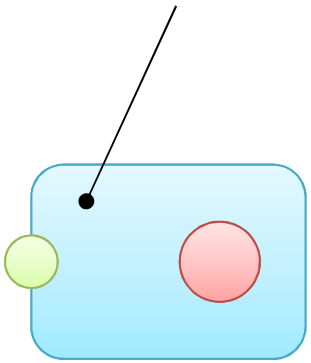
Konfiguracja XML

```
<bean id="textTemplate"
        class="lab.spring.TextTemplate"
        scope="prototype">
  ...
</bean>

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

Profile

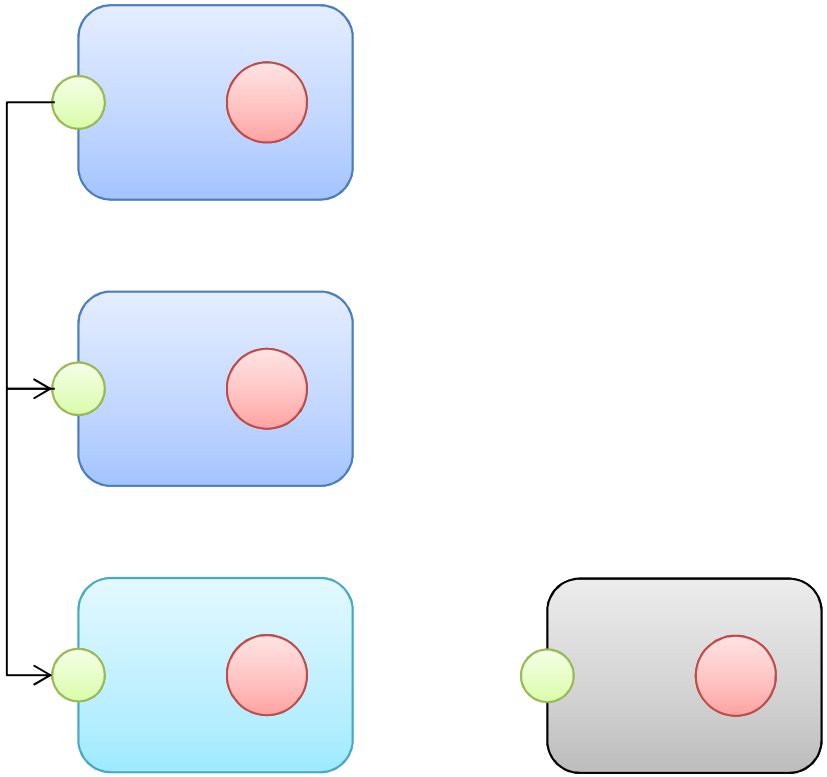


Wersja komponentu implementująca taki sam interfejs biznesowy co "niewygodny" komponent jednak tylko "udający" jego funkcjonalność

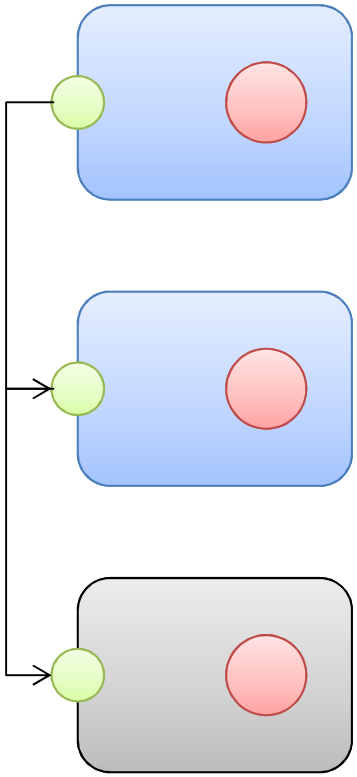


Potencjalnie "niewygodny" komponent w środowisku developerskim (albo innym) np. element systemu wysyłający dużą ilość maili.

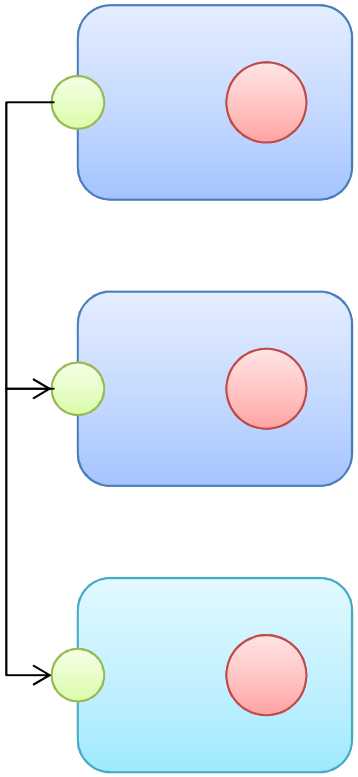
Wersja dla środowiska deweloperskiego



Co chcemy osiągnąć



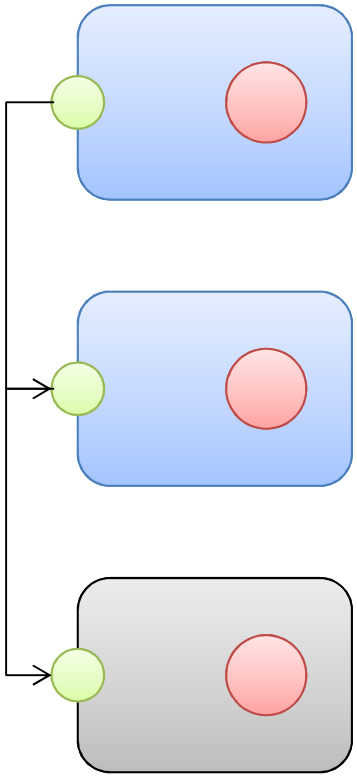
środowisko
"produkcyjne"



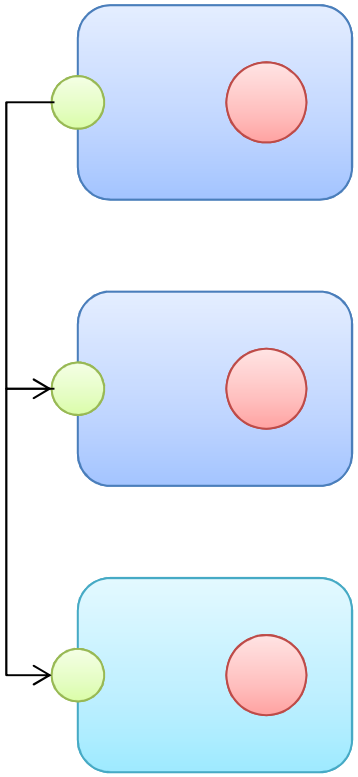
środowisko
programistyczne

VAVATECH

Profile



production



development

Profile

```
<beans profile="development">
    <bean id="dataSource"
          class="org.apache.commons.dbcp.BasicDataSource">

    </bean>
</beans>

<beans profile="production">
    <jee:jndi-lookup id="dataSource"
                     jndi-name="java:comp/env/jdbc/datasource"/>
</beans>
```

Użycie profili

```
GenericXmlApplicationContext ctx = new
GenericXmlApplicationContext();
ctx.getEnvironment().setActiveProfiles("profile1, profile2");
ctx.load("classpath:*-config.xml");
ctx.refresh();
```

```
-Dspring.profiles.active=profile1, profile2
```

Użycie profili

```
GenericXmlApplicationContext ctx = new
GenericXmlApplicationContext();
ctx.getEnvironment().setActiveProfiles("profile1, profile2");
ctx.load("classpath:*-config.xml");
ctx.refresh();
```

```
-Dspring.profiles.active=profile1, profile2
```

Spring Expression Language

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

SpEL – przegląd możliwości

- Wyrażenia dosłowne (literalne)
- Operatory logiczne
- Wyrażenia regularne
- Porównywanie klas
- Dostęp do własności, list, tablic, słowników
- Dostęp do komponentów
- 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']

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

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

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);
```

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.

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)

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

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

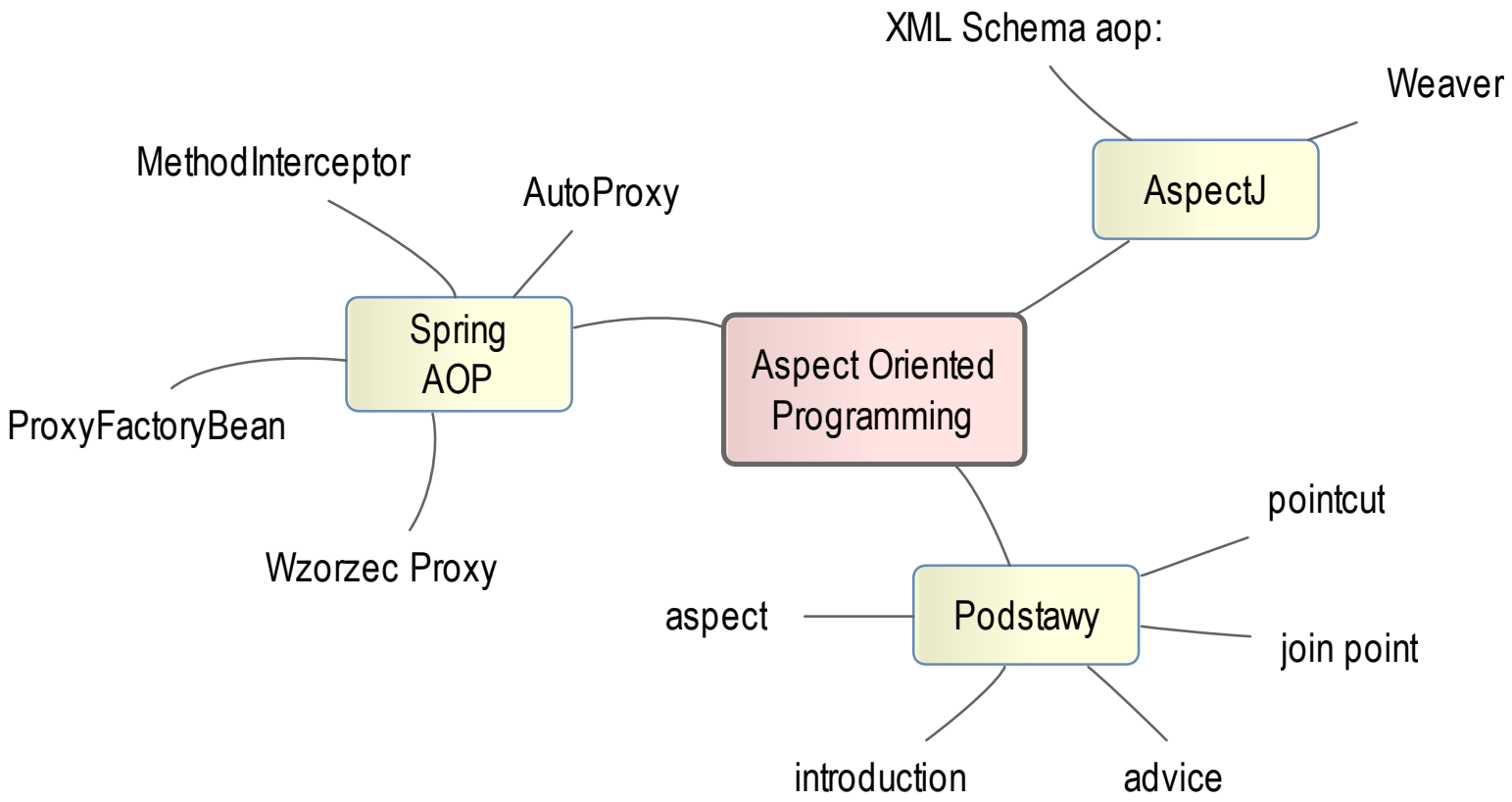
```
String randomPhrase =
    parser.parseExpression("random number is #{T(java.lang.Math).random()}",
                    new TemplateParserContext()).getValue(String.class);


public class TemplateParserContext implements ParserContext {

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

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

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

VAVATECH

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

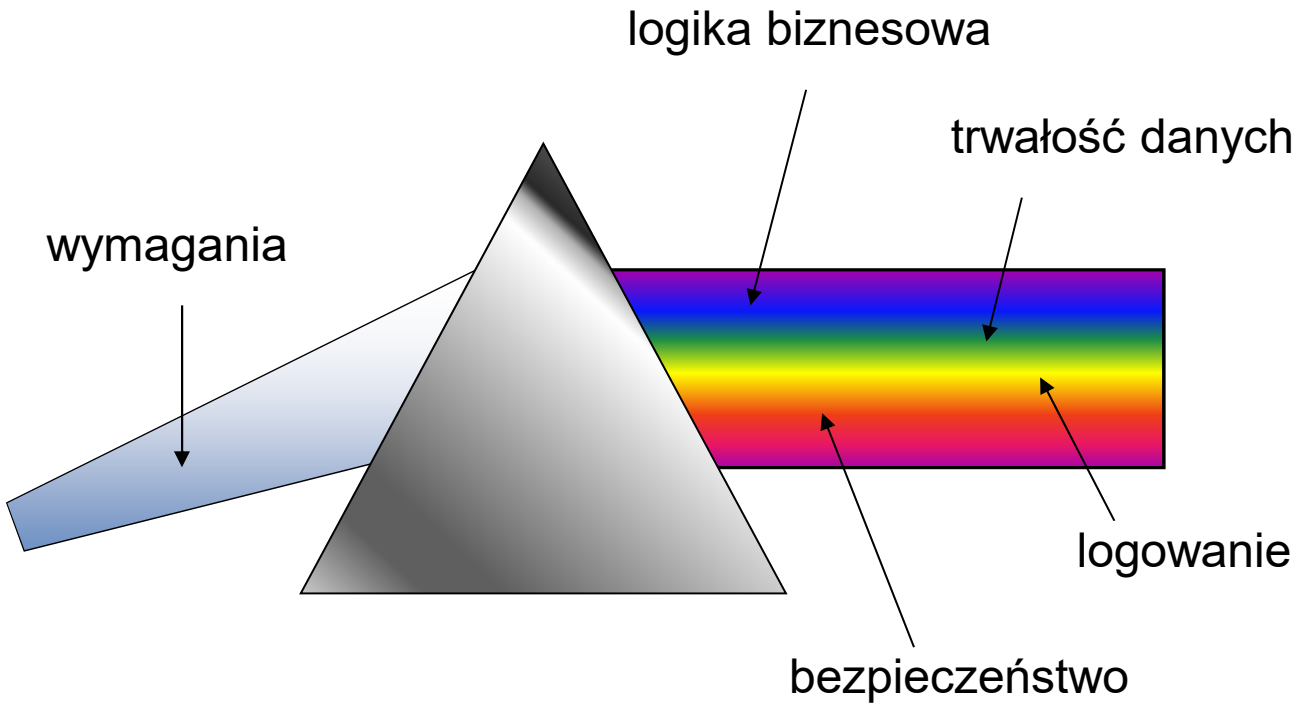
AOP

- 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).

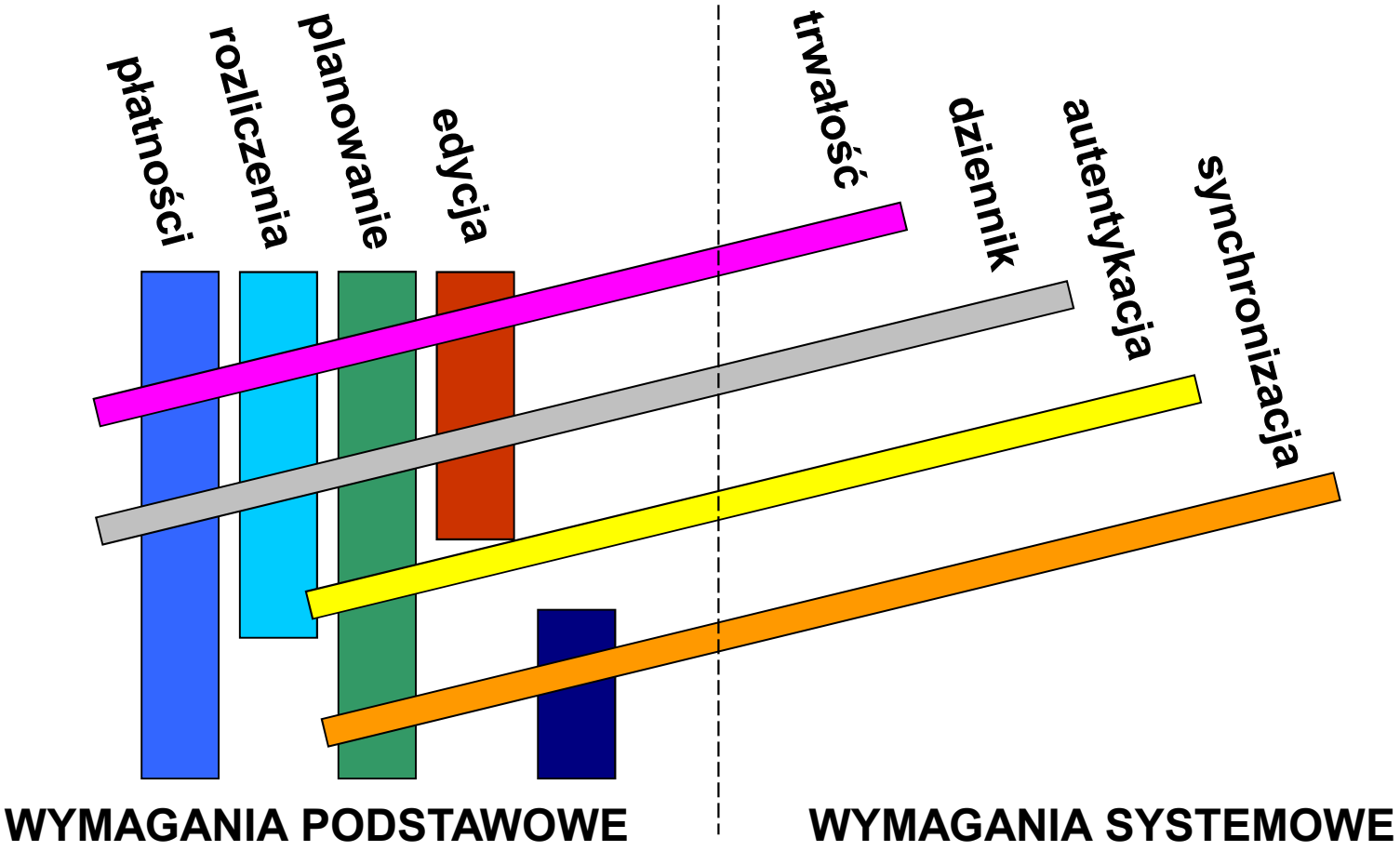
Modularyzacja

- 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.

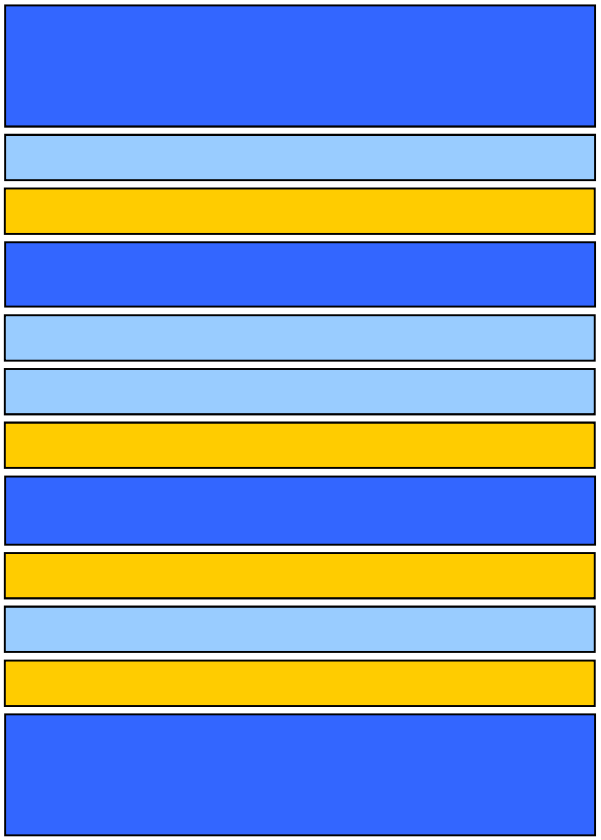
Wymagania klienta



Podział wymagań



Modularyzacja kodu



PROGRAM OBIEKTOWY

PROGRAM ASPEKTOWY

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

Punkt cięcia

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

Porada

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

Wprowadzenie

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

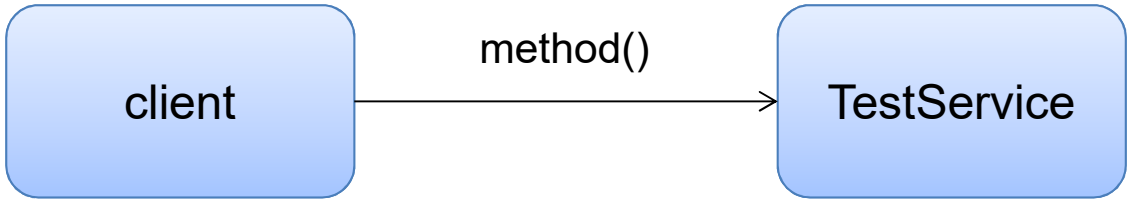
Aspekt

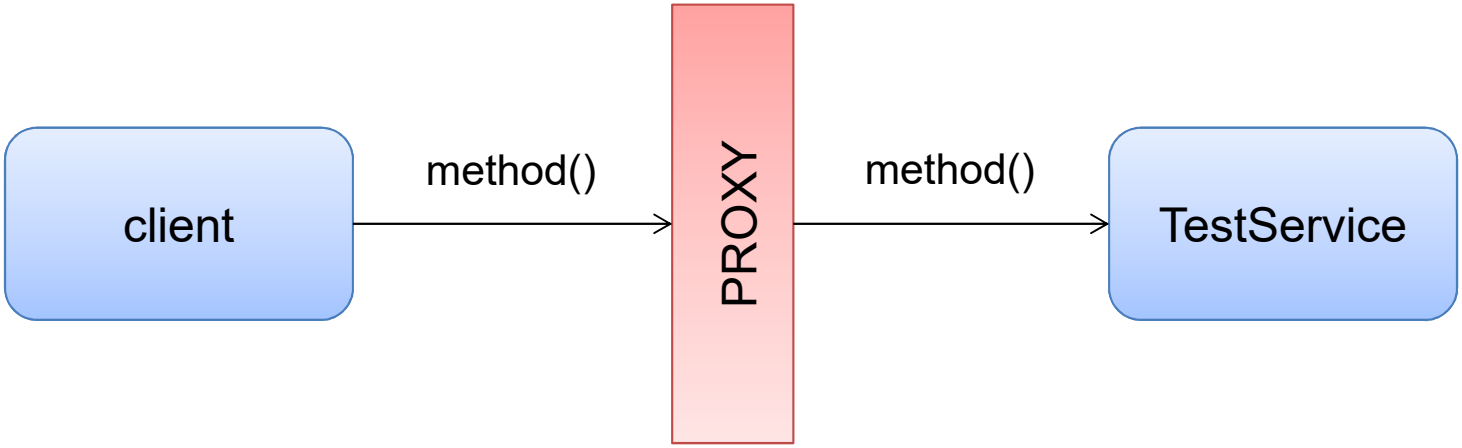
- **Aspekt** (ang. aspect) to kombinacja porad i punktów cięcia. Definiuje jaka logika ma zostać dołączona do aplikacji i gdzie ma zostać ona wywołana.

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 {
  private String data;

  public String getData() {
    return this.data;
  }

  public void setData(String data) {
    this.data=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 = context.getBean(TestService.class);
  service.setData("DATA");
}
```

JDK Proxy

```
class LoggingInvocationHandler implements InvocationHandler {

  private Object targetObject;

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

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

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 interface MethodBeforeAdvice extends BeforeAdvice {

        void before(Method m, Object[] args, Object target)
                                        throws Throwable;

}
```

org.springframework.aop.ThrowsAdvice

```
public classRemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex)
        throws Throwable {


    }

}
```

afterThrowing([Method, args, target], subclassOfThrowable)

org.springframework.aop.AfterReturningAdvice

```
public interface AfterReturningAdvice extends Advice {

        void afterReturning(Object returnValue, Method m,
                            Object[] args, Object target)
                throws Throwable;

}
```

org.aopalliance.intercept.MethodInterceptor

```
public class InvokeInterceptor implements MethodInterceptor {

        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="auditAdvisor"
        class="org.springframework.aop.support.DefaultPointcutAdvisor">
        <property name="advice">
        <ref local="threadLocalAuditAdvice" />
        </property>
        <property name="pointcut">
                <bean
                class="org.springframework.aop.support.JdkRegexpMethodPointcut">
                <property name="pattern">
                <value>^[a-zA-Z0-9.]+Service.*.*</value>
                </property>
                </bean>
        </property>
</bean>
 <bean id="autoProxyCreator"
    class="org.springframework.aop.framework.
                autoproxy.DefaultAdvisorAutoProxyCreator"/>
```

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 {
        …
        …
        }
}
```

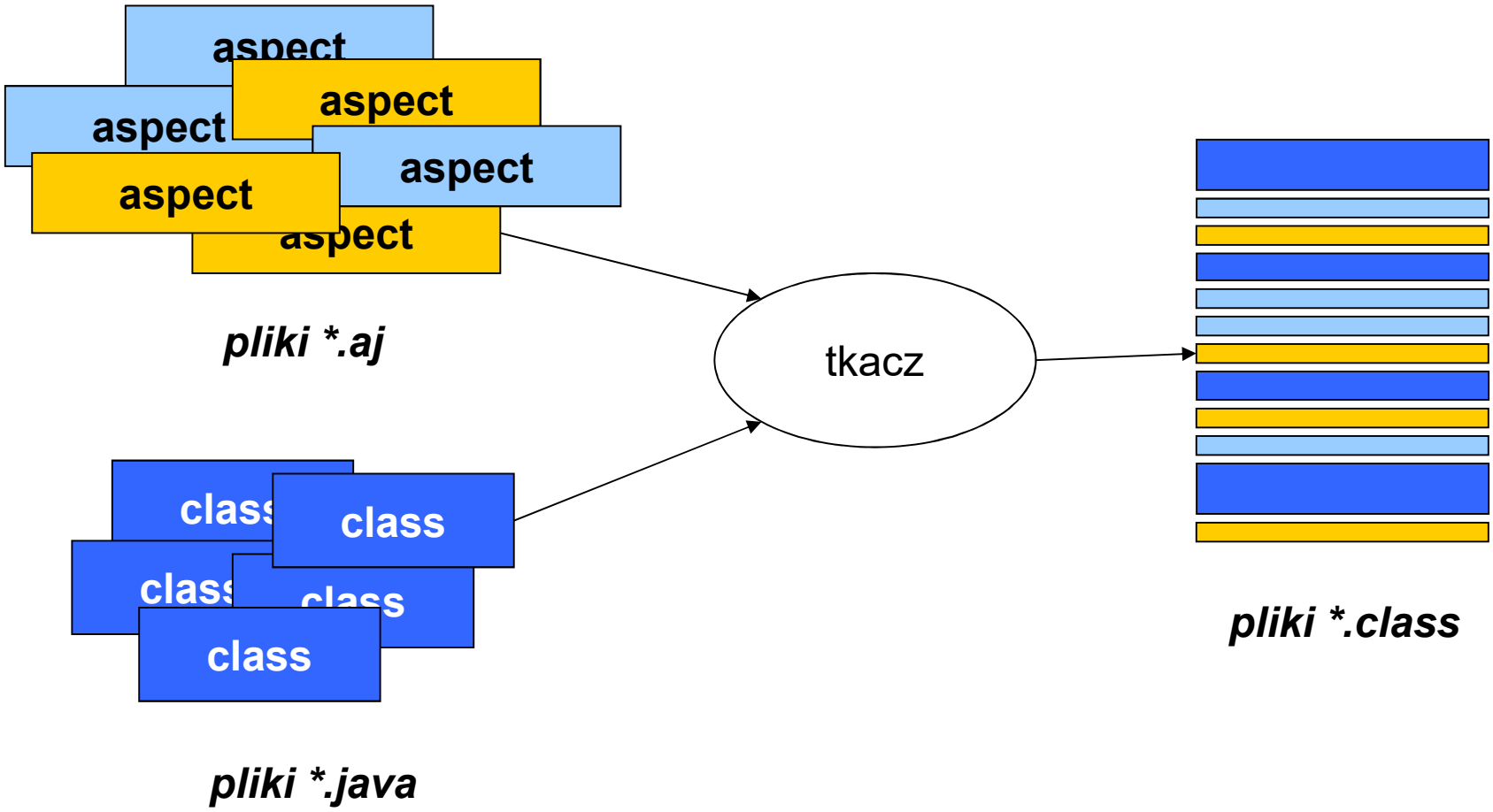
```
<bean id="orderedMonitorAdvice" class="OrderedMonitorAdvice">
        <property name="…" value="…"/>
</bean>

<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"/>
```

VAVATECH

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

Weaver (AspectJ)



@AspectJ

```
<aop:aspectj-autoproxy/>

<bean id="myAspect"
      class="org.xyz.NotVeryUsefulAspect">
</bean>
```

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

Wykorzystanie AspectJ

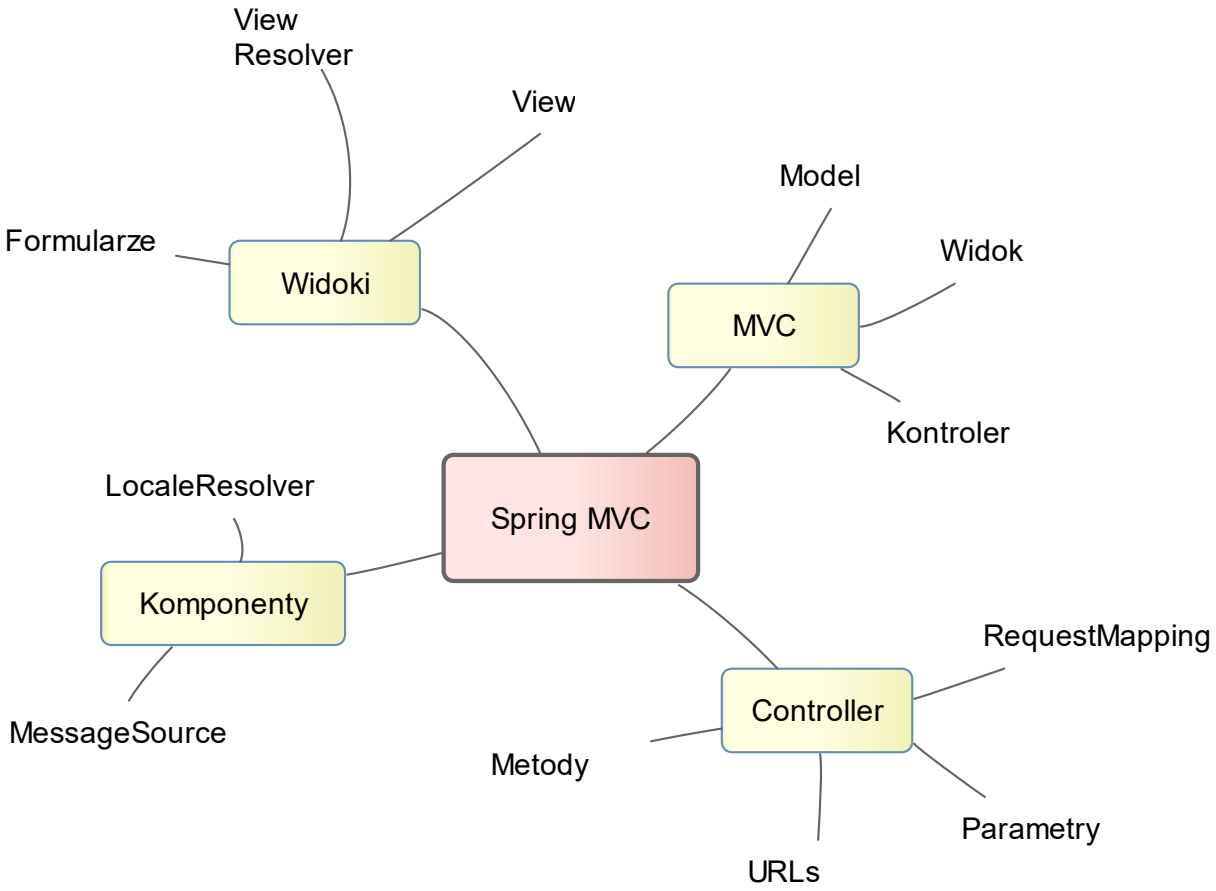
- Korzystanie z pełnych możliwości AspectJ wymaga kompilacji aplikacji za pomocą kompilatora ajc lub użycie tzw. load-time weaving czyli kompilacji aspektów w czasie ładowania klasy do jvm.

VAVATECH

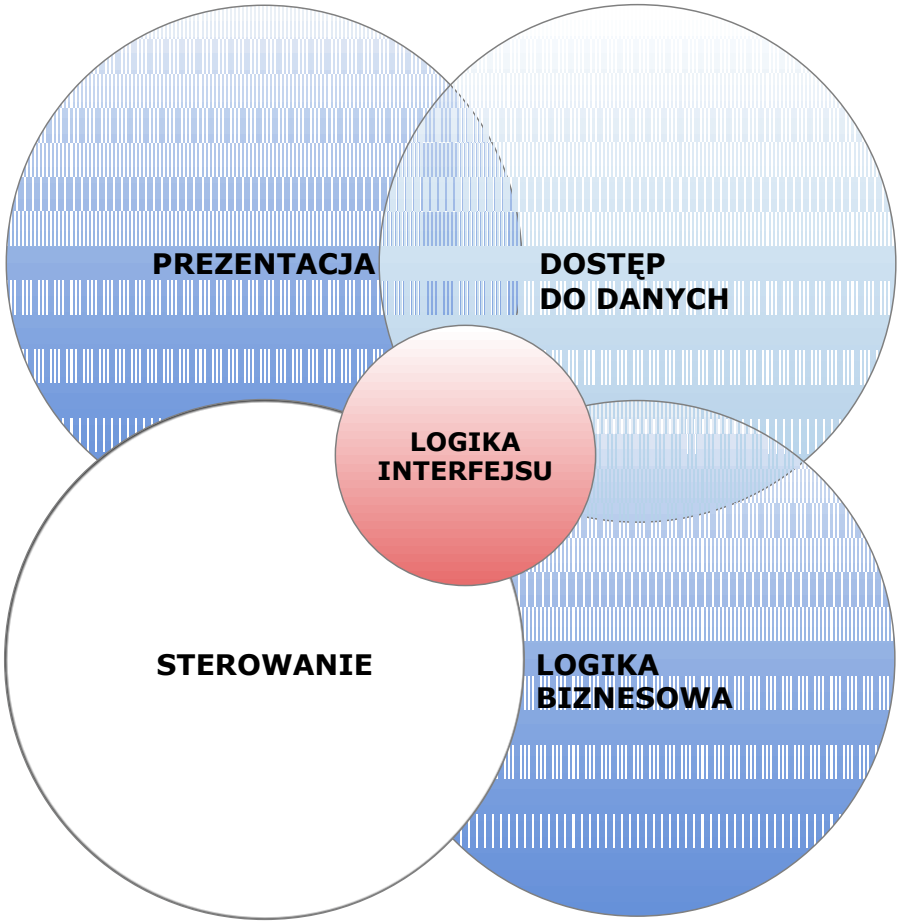
Spring MVC

Tworzenie aplikacji web

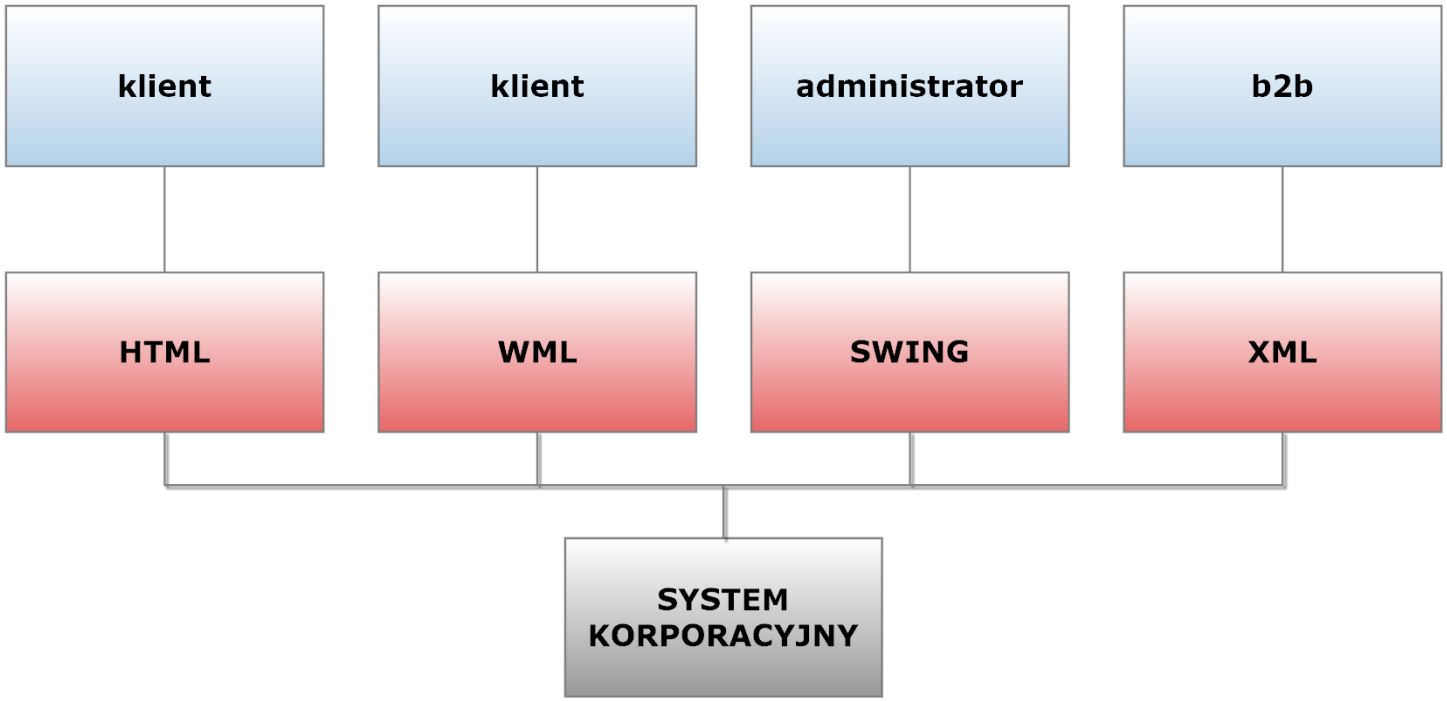
Agenda


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

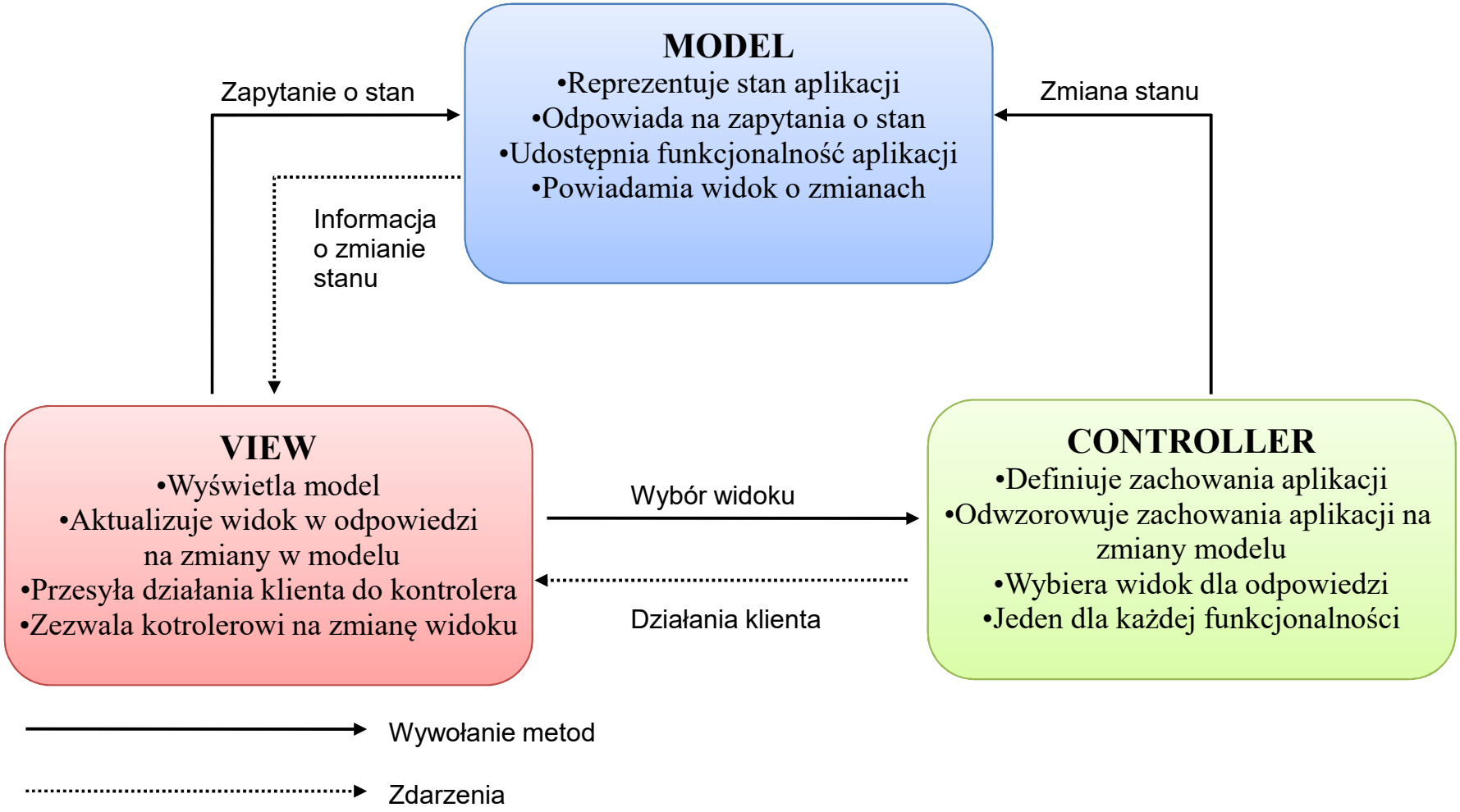
Podejście "tradycyjne"



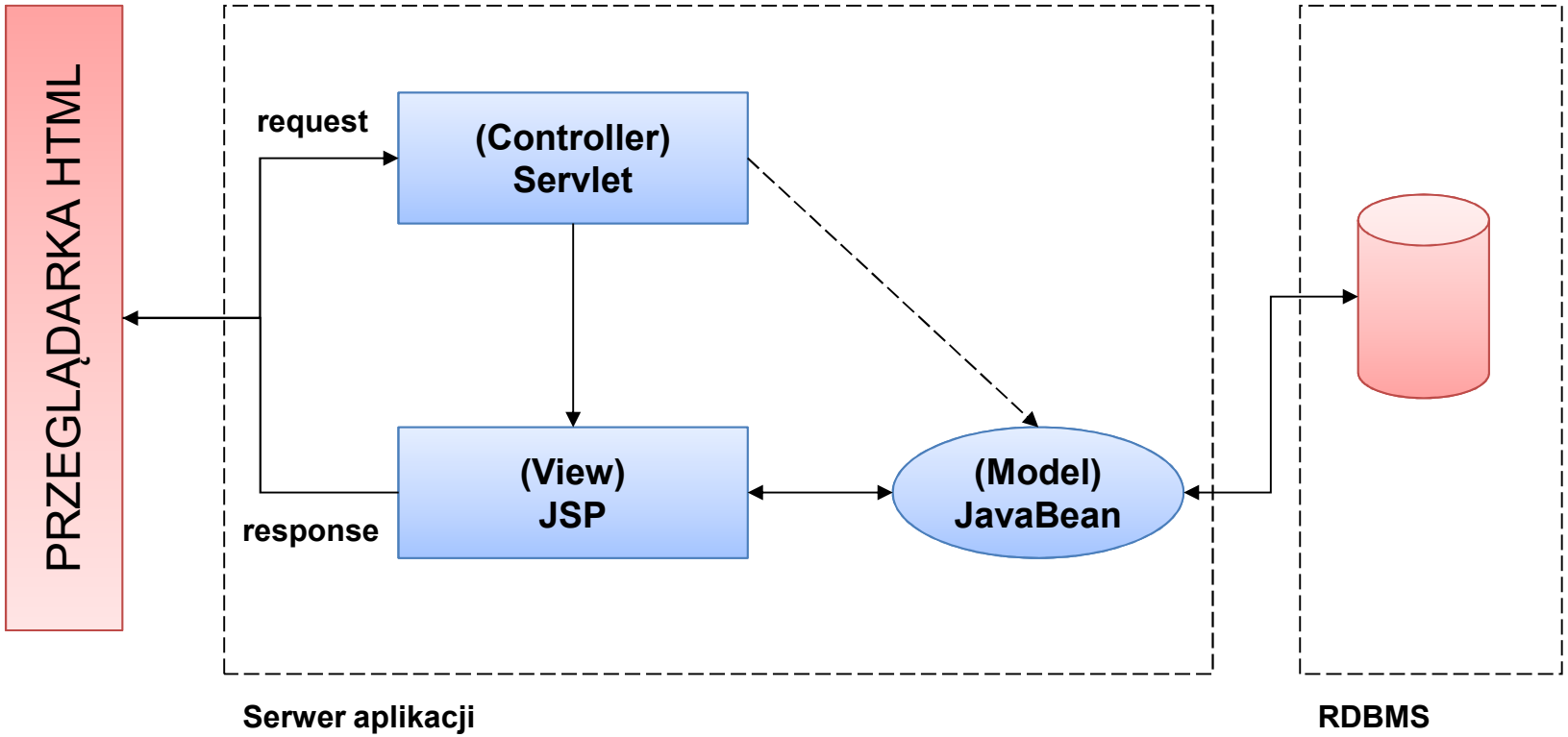
Problem do rozwiązania



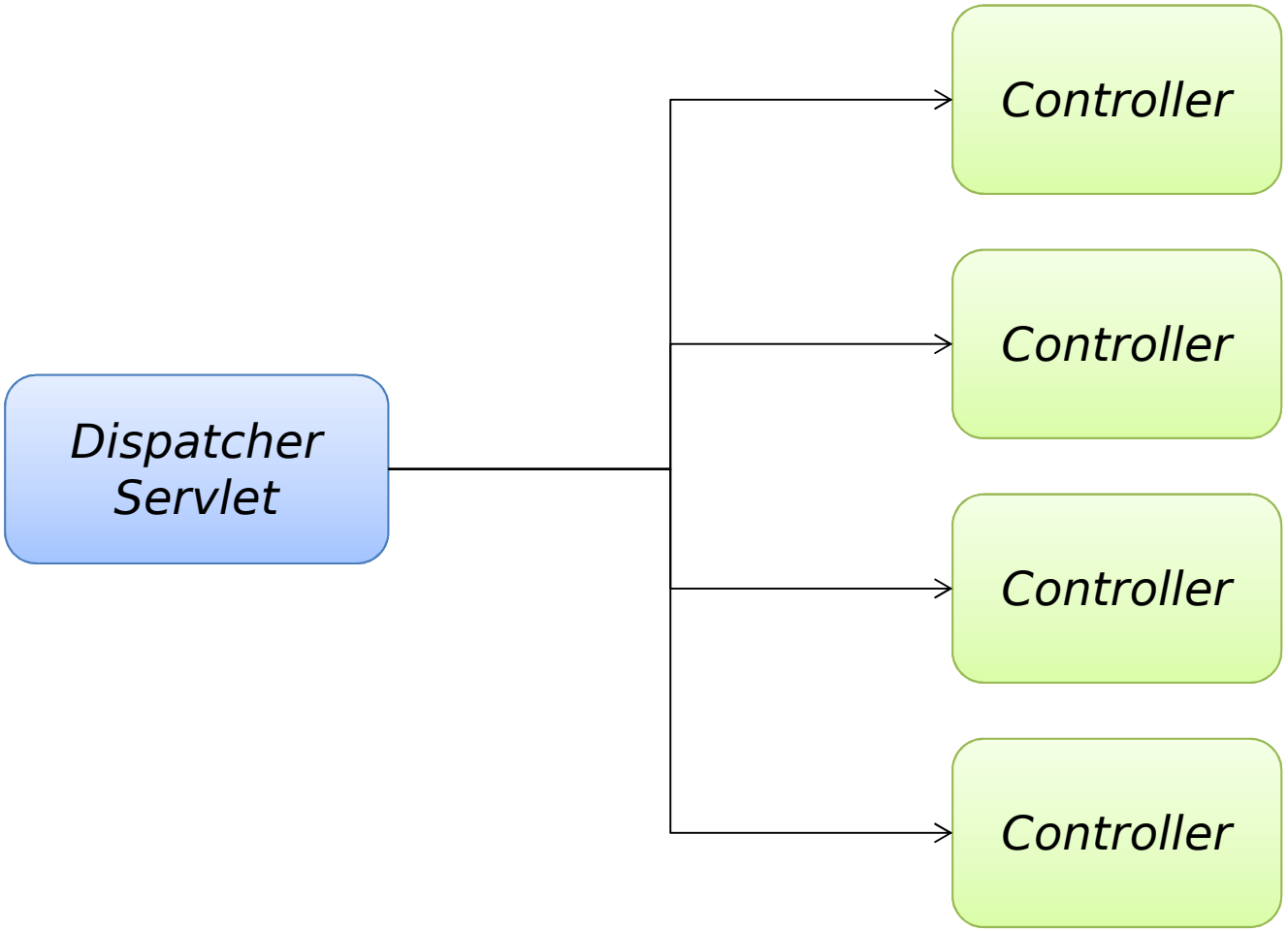
Model View Controller



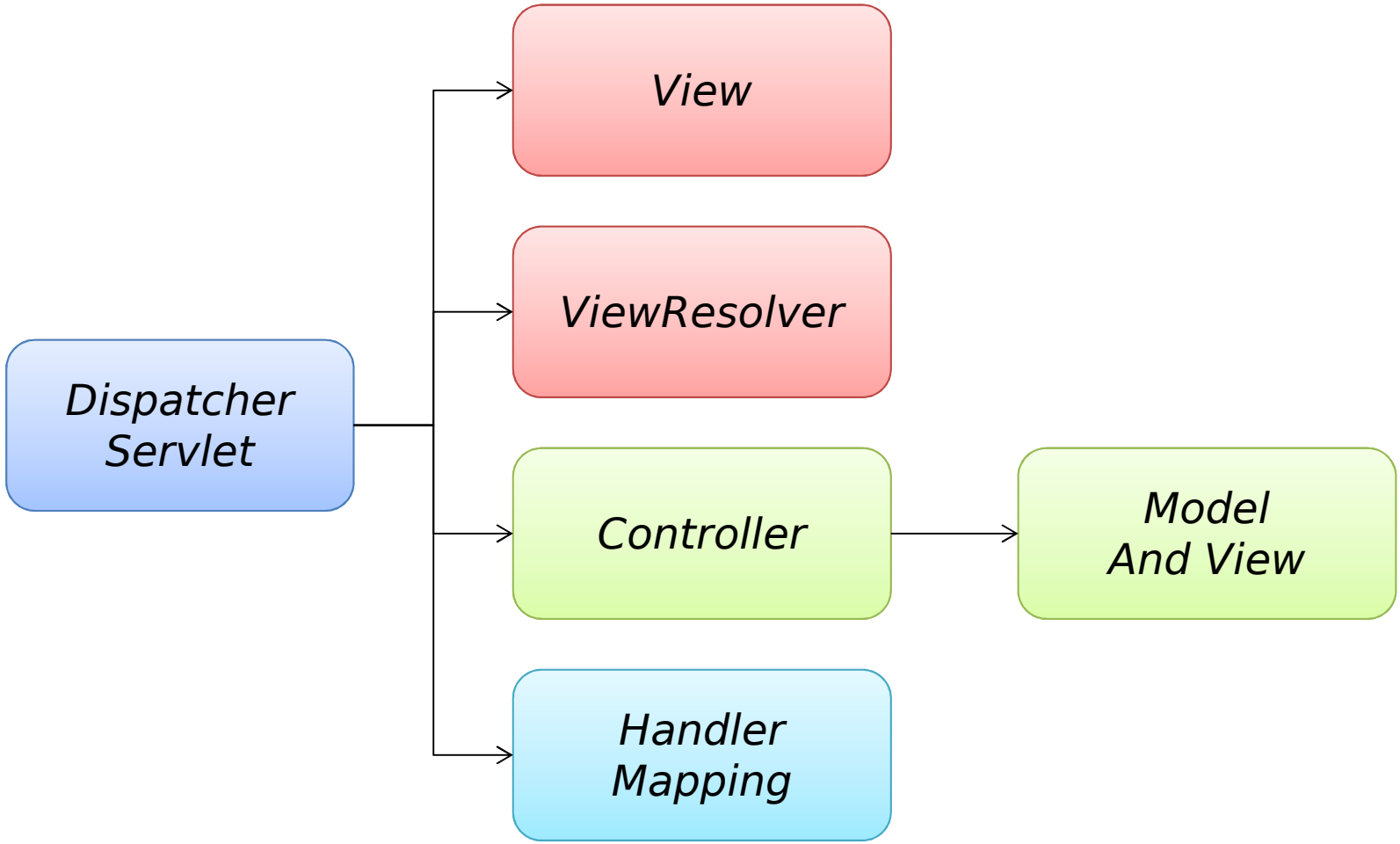
Standardowe MVC w środowisku Java EE



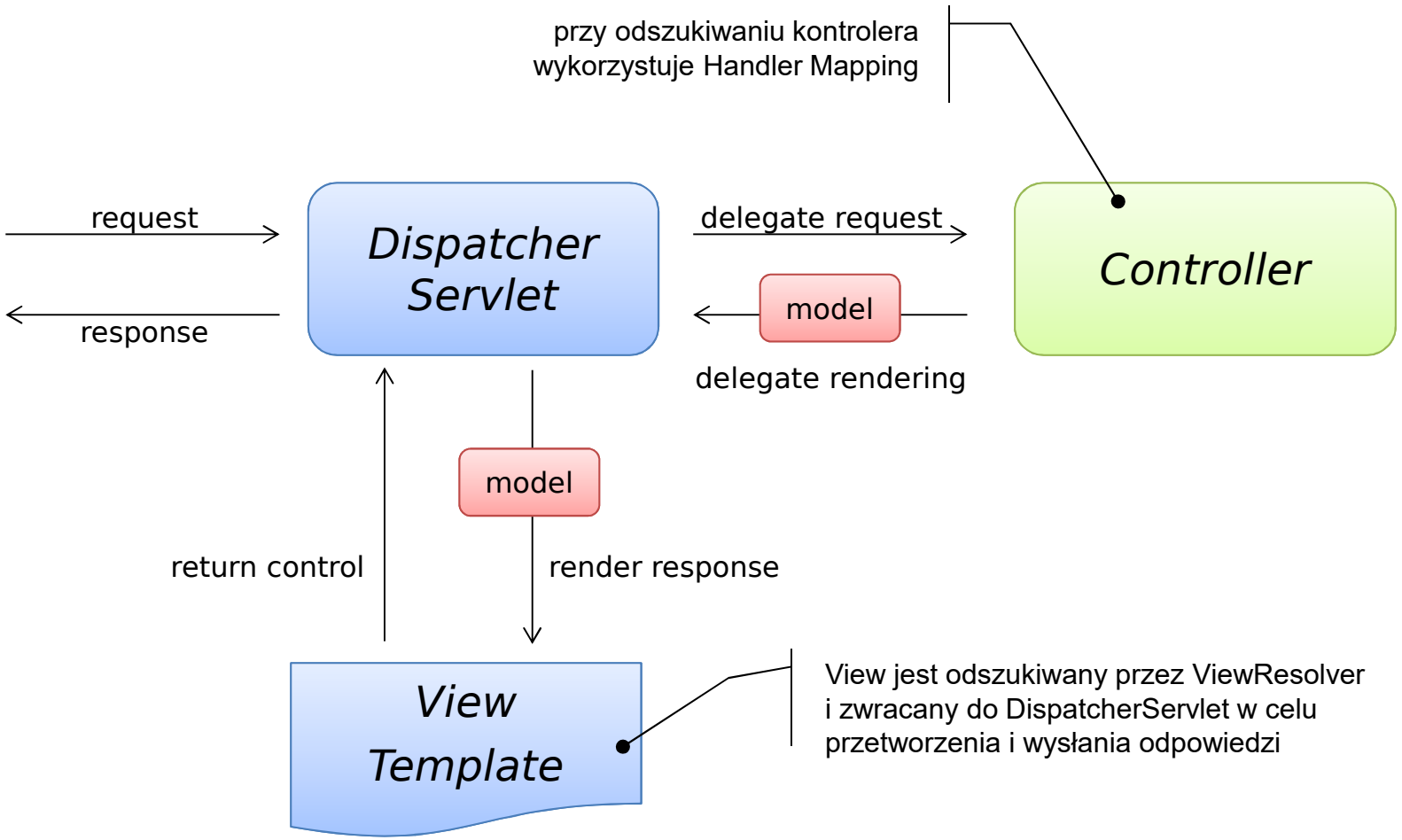
Rozwiązanie architektoniczne



Komponenty Spring MVC



Przepływ danych w Spring MVC



Konfiguracja Dispatcher Servlet

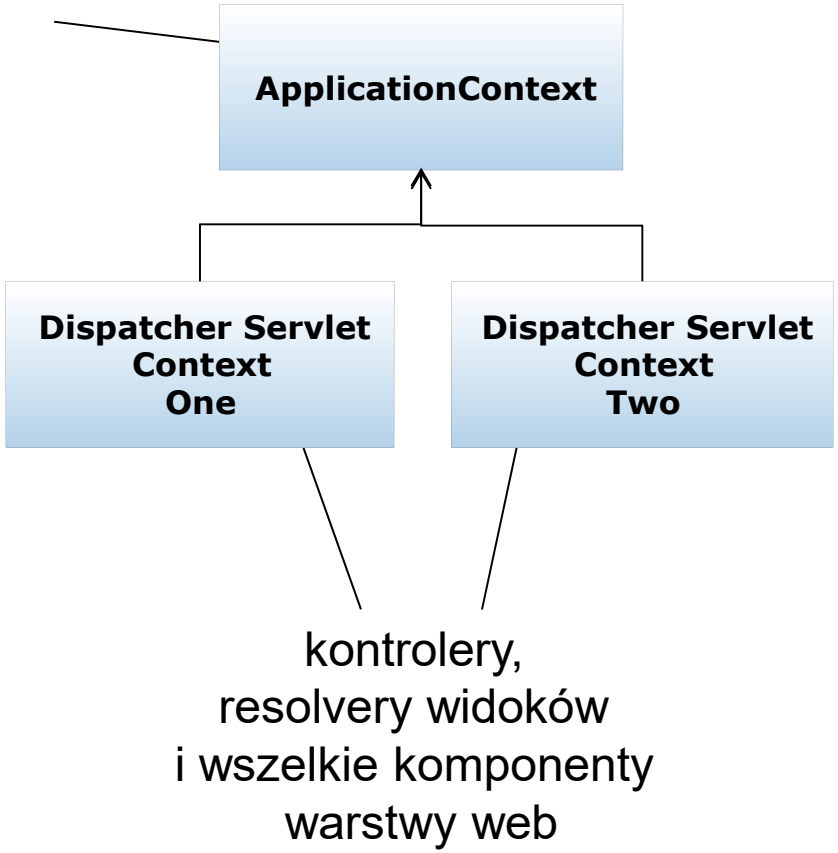
```
<servlet>
        <servlet-name>spring-mvc</servlet-name>
        <servlet-class>
                org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
</servlet>

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

Servlet API 3.0+

```
import org.springframework.web.WebApplicationInitializer;

public class MyWebApplicationInitializer
                    implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext container) {
    XmlWebApplicationContext appContext =
                    new XmlWebApplicationContext();
    appContext.setConfigLocation(
                    "/WEB-INF/spring/dispatcher-config.xml");
    ServletRegistration.Dynamic registration =
          container.addServlet("dispatcher",
                    new DispatcherServlet(appContext));
    registration.setLoadOnStartup(1);
    registration.addMapping("/");
    }
}
```

Konfiguracja kontekstu aplikacji

komponenty warstwy logiki biznesowej, warstwy danych



kontrolery, resolvery widoków i wszelkie komponenty warstwy web

Konfiguracja kontekstu aplikacji (web.xml)

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

Nie trzeba definiować lokalizacji w przypadku gdy plik konfiguracyjny jest jeden i znajduje się w domyślnej lokalizacji /WEB-INF/applicationContext.xml

Kontekst servletu

```
<servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
                org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
        <param-name>contextConfigLocation</param-name>
                <param-value>
                        /WEB-INF/spring/mvc-config.xml
                </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
</servlet>
```

Nie trzeba definiować lokalizacji w przypadku
gdy plik konfiguracyjny korzysta z konwencji nazewniczej
/WEB-INF/**dispatcher**-servlet.xml

Kontrolery w Spring 3.0

- W pierwszych wersjach Spring MVC instniała obiektowa hierarchia kontrolerów. 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 kontrolerów

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
                http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:mvc="http://www.springframework.org/schema/mvc">

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

</beans>
```

Elementy konfiguracji Spring MVC

- mvc:annotation-driven
  - Uruchamia wsparcie dla konwersji i formatowania za pomocą adnotacji
  - Uruchamia walidację JSR-303
  - Umożliwia podpięcie własnego ConversionService

```
@Configuration
@EnableWebMvc
public class WebConfig {

}
```

mvc:view-controller

- mvc:view-controller
  - Definiuje kontroler, który zawsze przekierowuje do wskazanego widoku

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

Dostosowanie środowiska Spring MVC

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    protected void addFormatters(FormatterRegistry registry) {

    }
    @Override
    public void configureMessageConverters(
                List<HttpMessageConverter<?>> converters) {

    }
}
```

Konwerteryi formatery

```
<mvc:annotation-driven conversion-service="conversionService">
        <mvc:message-converters>
        <bean class="org.example.MyHttpMessageConverter"/>
        <bean class="org.example.MyOtherHttpMessageConverter"/>
        </mvc:message-converters>
</mvc:annotation-driven>

<bean id="conversionService"
        class="org.springframework.format.support.
                FormattingConversionServiceFactoryBean">
    <property name="formatters">
    <list>
    <bean class="org.example.MyFormatter"/>
    <bean class="org.example.MyOtherFormatter"/>
    </list>
    </property>
</bean>
```

Interceptory

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(newLocaleInterceptor());
        registry.addInterceptor(
                new ThemeInterceptor()).
                addPathPatterns("/**").
                excludePathPatterns("/admin/**");
        registry.addInterceptor(
                new SecurityInterceptor()).
                addPathPatterns("/secure/*");
    }
}
```

Interceptory w XML

```
<mvc:interceptors>
    <bean class="org.springframework.web.servlet.i18n.
        LocaleChangeInterceptor"/>
    <mvc:interceptor>
    <mapping path="/**"/>
    <exclude-mapping path="/admin/**"/>
    <bean class="org.springframework.web.servlet.theme.
        ThemeChangeInterceptor"/>
    </mvc:interceptor>
    <mvc:interceptor>
    <mapping path="/secure/*"/>
    <bean class="org.example.SecurityInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

VAVATECH

Klasa interceptora

- 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

Przykład interceptora

```
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;
    }
```

Przykład interceptora

```
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/page.html");
        return false;
    }
  }
}
```

Przykład konfiguracji

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

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

ContentNegotiation

```
@Configuration
@EnableWebMvc
public class WebConfig extendsWebMvcConfigurerAdapter {

    @Override
    public void configureContentNegotiation(
                    ContentNegotiationConfigurer configurer) {
        configurer.favorPathExtension(false).
                        favorParameter(true);
    }
}
```

ContentNegotiation w XML+

```
<mvc:annotation-driven
        content-negotiation-manager=
                "contentNegotiationManager"/>
<bean id="contentNegotiationManager"
        class="org.springframework.web.accept.
                ContentNegotiationManagerFactoryBean">
    <property name="favorPathExtension" value="false"/>
    <property name="favorParameter" value="true"/>
    <property name="mediaTypes">
    <value>
        json=application/json
        xml=application/xml
    </value>
    </property>
</bean>
```

View Controller

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(
                    ViewControllerRegistry registry) {
        registry.addViewController("/")
                    .setViewName("home");
    }

}
```

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

Resources

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(
                    ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**").
        addResourceLocations("/publicresources/").
        setCachePeriod(31556926);
    }
}
```

Resources c.d.

```
@EnableWebMvc
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(
        ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
        .addResourceLocations(
                "/", "classpath:/META-INF/public-web-resources/");
    }

}
```

Resources w XML

```
<mvc:resources
        mapping="/resources/**"
        location="/public-resources/"
        cacheperiod="31556926"
/>

<mvc:resources
        mapping="/resources/**"
        location="/,
        classpath:/META-INF/public-web-resources/"
/>
```

@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;
    }

}
```

Mapowanie klasy i metody

```
@Controller
@RequestMapping("/hello")
public class HelloWorldController {

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

}
```

Docelowy URL metody to */hello/helloWorld*

@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.

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 metod kontrolera

- Nie ma sprecyzowanych parametrów metod kontrolera.
- Programista ma możliwość określania jakie parametry go interesują i zdefiniować je w sygnaturze metody.
- Spring rozpozna je i odpowiednio wywoła metodę.
- Można wybierać z określonego zakresu typów parametrów.

Parametry metod kontrolera

- 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, @CookieValue, @MatrixVariable
- java.util.Map
- Command Object

@RequestParam

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

@RequestHeader

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

    ...
    ...

}
```

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) {
 // ...
}
```

Matrix Variables

- Rodzaj parametrów w postaci klucz wartość zawartych w ścieżkę wywołania.
- Określone są za pomocą dokumentu RFC 3986.
- Przykładowo
  - /cars;color=red;year=2012
  - /cars; color=red;color=green;color=blue

Obsługa Matrix Variables

- ***GET /pets/42;q=11;r=22***

```
@RequestMapping(value = "/pets/{petId}",
                        method = RequestMethod.GET)
public void findPet(@PathVariableString petId,
                        @MatrixVariable intq) {
        // petId == 42
        // q == 11
}
```

- ***GET /owners/42;q=11/pets/21;q=22***

```
@RequestMapping(value = "/owners/{ownerId}/pets/{petId}",
                        method = RequestMethod.GET)
public void findPet(
    @MatrixVariable(value="q", pathVar="ownerId") intq1,
    @MatrixVariable(value="q", pathVar="petId") intq2) {
        // q1 == 11
        // q2 == 22
}
```

Wielokrotne wartości

- ***GET /owners/42;q=11;r=12/pets/21;q=22;s=23***

```
@RequestMapping(value = "/owners/{ownerId}/pets/{petId}",
                        method = RequestMethod.GET)
public void findPet(
            @MatrixVariable Map<String, String> matrixVars,
            @MatrixVariable(pathVar="petId"")
                                    Map<String, String> petMatrixVars) {

    // matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]
    // petMatrixVars: ["q" : 11, "s" : 23]

}
```

VAVATECH

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

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

Wybieranie widoków

- Widok jest określany przez ViewResolver na podstawie url metody kontrolera lub identyfikatora widoku zwracanego przez tą metodę.
- Dzięki identyfikatorowi a nie bezpośredniego odwołania się do lokalizacji widoku możliwe jest dowolne mapowanie widoku na identyfikator.

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

Konfiguracja kontenera

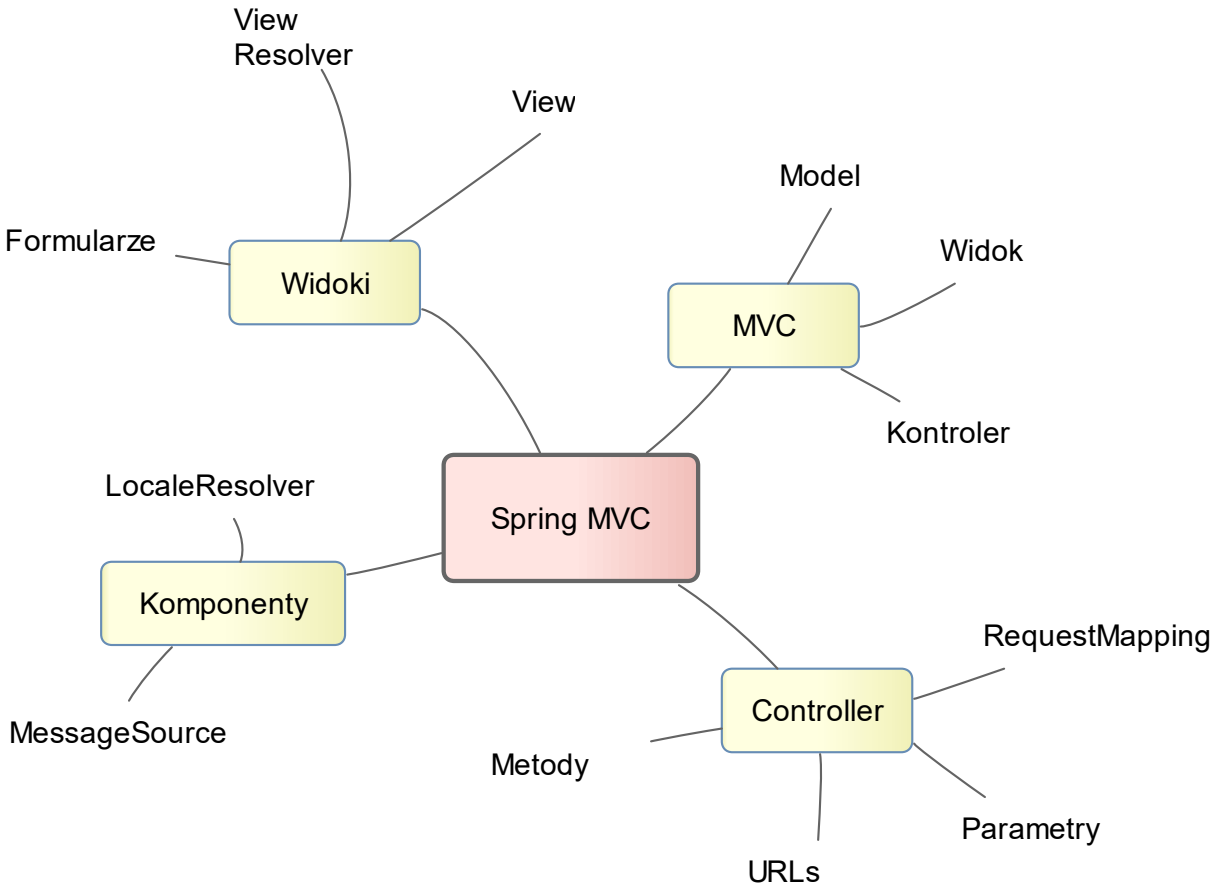
```
<bean class="org.springframework.web.servlet.view.
                    InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
</bean>
```

VAVATECH

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

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.

Interfejsy

```
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:

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

  - domyślny plik zasobów: views.properties

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

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

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

Przykładowa konfiguracja

```
<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>
```

c.d.

```
<property name="defaultViews">
    <list>
      <bean class="org.springframework.web.servlet.view.json.
                                MappingJacksonJsonView" />
    </list>
  </property>
</bean>
```

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

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

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"/>
```

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>
```

Formularz

```
<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

```
@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";
                }
}
```

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

@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

Przykład

```
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

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

VAVATECH

Techniki renderowania widoków i tworzenia formularzy

Biblioteki znaczników JSP

Walidacja i konwersja danych

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)

Biblioteki tagów Spring

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

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

Biblioteka tagów dla formularzy

- Zawartość *http://www.springframework.org/tags/form* :
  - *form*
  - *input*
  - *checkbox, checkboxes*
  - *radiobutton, radiobuttons*
  - *password*
  - *select*
  - *option, options*
  - *textarea*
  - *hidden*
  - *errors*

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

Przykład formularza

```
<%@ 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

Errors przykład

```
<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>
```

Biblioteka innych znaczników

- Zawartość *http://www.springframework.org/tags*
  - *bind*
  - *escapeBody*
  - *hasBindErrors*
  - *htmlEscape*
  - *message*
  - *nestedPath*
  - *theme*
  - *transform*
  - *url*
  - *eval*

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="..."/>

Walidacja za pomocą adnotacji JSR-303

```
public class Person {

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

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

}
```

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

BeanWrapper

- BeanWrapper – klasa ułatwiająca pracę na obiektach typu JavaBean.
- Pozwala na działania w oparciu o nazwy właściwości klas.
- Umożliwia również działania na kolekcjach, listach, właściwościach zagnieżdżonych.

Przykładowa klasa

```
public class Company{

  private String name;

  private String city;

  private Address address;

  // ... gettery, settery itp.

}
```

BeanWrapperImpl

```
Company company = new Company();

BeanWrapper beanWrapper =
        new BeanWrapperImpl(company);

beanWrapper.setPropertyValue("name", "ACME");
```

Dostęp do właściwości

Właściwość	
name	Dostęp do wartości poprzez getName() lub setName()
address.city	Dostęp do wartości poprzez getAddress().getCity() lub getAddress().setCity()
products[1]	Dostęp do indeksowanej wartości listy, tablicy lub kolekcji.
products[PRODUCT_NAME]	Dostęp do mapy poprzez wartość klucza.

PropertyValue

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

PropertyValue value = new PropertyValue("name", "ACME");
company.setPropertyValue(value);

BeanWrapper address= BeanWrapperImpl(new Address());
address.setPropertyValue("city", "Warszawa");

company.setPropertyValue("address",
address.getWrappedInstance());
```

PropertyEditor

- Ze względu na konfigurację komponentów Spring lub odbiór parametrów HTTP za pomocą java.lang.String potrzebny jest mechanizm konwersji pomiędzy wartością tekstową i konkretnym typem.
- Podstawowy mechanizm oparty jest o klasę

  java.beans.PropertyEditor

Wbudowane typy PropertyEditor

- ByteArrayPropertyEditor
- ClassEditor
- CustomBooleanEditor
- CustomCollectionEditor
- CustomDateEditor
- CustomNumberEditor
- FileEditor
- InputStreamEditor
- LocaleEditor
- PatternEditor
- PropertiesEditor
- StringTrimmerEditor
- URLEditor

Własny PropertyEditor

- PropertyEditor podaje się wg specyfikacji JavaBeans za pomocą odpowiedniej metody.
- Wymaga to sporej dodatkowej pracy przy definiowaniu komponentów.
- Łatwiejszy sposób – wykorzystanie PropertyEditorSupport

PropertyEditorSupport

```
package lab.spring;

public class AddressTypeEditor
                extends PropertyEditorSupport {

        public void setAsText(String text) {
                setValue(new Address(...));
        }

}
```

Rejestracja Custom PropertyEditor

- PropertyEditor można zarejestrować w BeanWrapper lub w kontenerze IoC.

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
<property name="customEditors">
    <map>
    <entry key="lab.spring.Address"
        value=" lab.spring.AddressTypeEditor "/>
    </map>
</property>
</bean>
```

Converter

```
public interface Converter<S, T> {

        T convert(S source);

}
```

Przykład

```
public class AddressConverter
        implements Converter<String, Address> {

  @Override
  public Address convert(String source) {
          //parse source
          return new Address();
  }

}
```

Rejestracja

```
<bean id="conversionService"
            class="org.springframework.context.support.ConversionServiceFactoryBean">
   <property name="converters">
     <set>
       <bean class="lab.spring.AddressConverter"/>
     </set>
   </property>
</bean>
```

Użycie

```
<bean id="person" class="lab.spring.Company">
    <property name="address" value="Krakowska 10"/>
</bean>
```

Formatter

- W przypadku korzystania z UI i takich elementów jak Spring MVC przydatna jest funkcja konwersji obiektu do wartości tekstowych do wyświetlenia dla użytkownika.
- Do tego powstał Formatter.

Formatter

```
public interface Formatter<T>
        extends Printer<T>, Parser<T> {
}

public interface Printer<T> {
        String print(T fieldValue, Locale locale);
}

public interface Parser<T> {
        T parse(String clientValue, Locale locale)
                throws ParseException;
}
```

Wbudowane Formattery

- NumberFormatter
- CurrencyFormatter
- PercentFormatter
- DateFormatter

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 konwerterów i formatterów w MVC

```
<mvc:annotation-driven conversion-service="conversionService"/>

<bean id="conversionService"
        class="org.springframework.format.support.
                FormattingConversionServiceFactoryBean">
    <property name="converters">
        <set>
        <bean class="org.example.MyConverter"/>
        </set>
    </property>
    <property name="formatters">
        <set>
        <bean class="org.example.MyFormatter"/>
        </set>
    </property>
</bean>
```

Klasa kompany

```
public class Company {

    @Size(max = 100)
    private String name;

    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
    private Date foundDate;

    private String numberOfEmployees;

...
...

}
```

Przykład formularza z walidacją

```
<form:form action="form" method="post"
        commandName="company">
    <form:errors path="*"/>

    Nazwa <form:input path="name"/> <br/>
    Data założenia <form:input path="foundDate"/> <br/>
    <input type="submit" value="Zapisz"/>

</form:form>
```

Kontroler

```
@RequestMapping(value = "/form", method = RequestMethod.POST)
public void processForm(
        @ModelAttribute("company") @Valid Company company,
        BindingResult errors) {

}

@RequestMapping(value = "/form", method = RequestMethod.GET)
public void form(@ModelAttribute("company") Company company) {


}
```

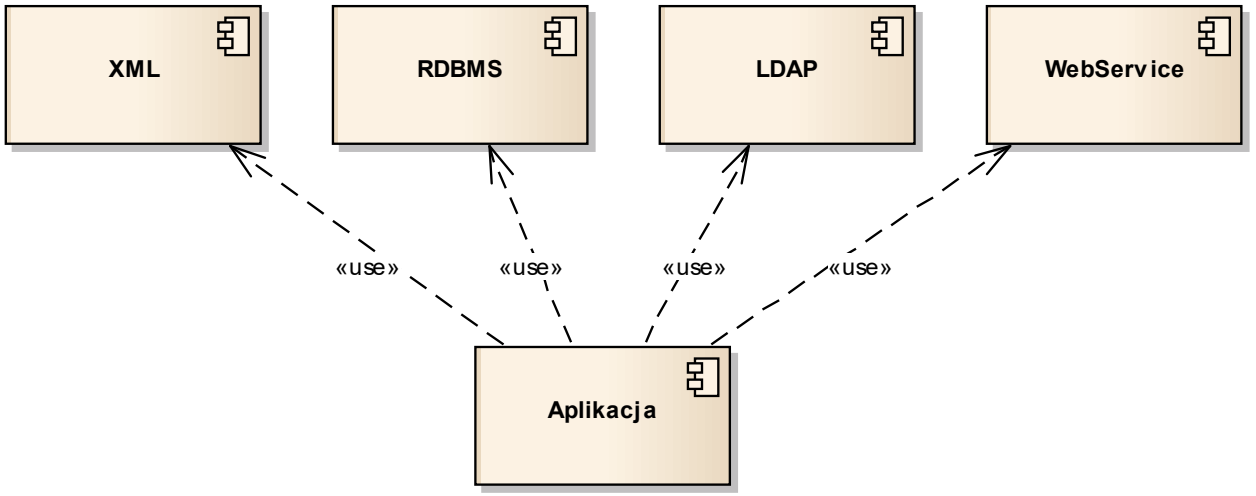
VAVATECH

Spring JPA

Wsparcie dla warstwy DAO
w Spring Framework

Problem

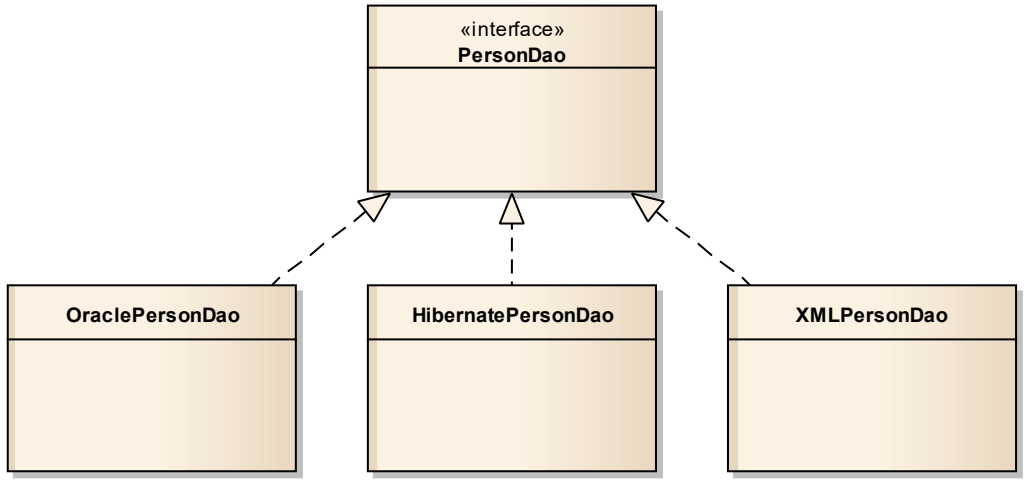
- Większość aplikacji biznesowych jako trwałych magazynów używa systemów zarządzania relacyjnymi bazami danych (RDBMS). Jednak dane biznesowe mogą znajdować się również w innych miejscach np zewnętrznych systemach mainframe, repozytoriach LDAP, obiektowych bazach danych, plikach.



VAVATECH

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



Wsparcie dla DAO w Spring

- Zestaw klas wspomagających tworzenie różnych implementacji DAO
- Hierarchia wyjątków ułatwiająca obsługę błędów
- Odciążenie programisty od wykonywania podstawowych i powtarzalnych operacji
  - otwarcie i zamknięcie połączenia i obiektów powiązanych
  - otwarcie Statement i PreparedStatement
  - realizacja pętli pobierającej dane
  - obsługa transakcji

Klasy DaoSupport

- JdbcTemplate/JdbcDaoSupport
- HibernateTemplate/HibernateDaoSupport
- JdoTemplate/JdoDaoSupport
- JpaTemplate/JpaDaoSupport
- SqlMapClientTemplate/SqlMapClientDaoSupport

Hierarchia wyjątków (wybór)

- DataAccessException
  - DataIntegrityViolationException
  - DuplicateKeyException
  - HibernateJdbcException
  - HibernateQueryException
  - InvalidResultSetAccessException
  - IncorrectResultSetColumnCountException
  - OptimisticLockingFailureException
  - PessimisticLockingFailureException
  - QueryTimeoutException
  - UncategorizedDataAccessException
  - UncategorizedSQLException

PersonDao

```
public interface CompanyDao {

    public Company get(Long id);

    public List<Company> selectAll();

    public void save(Company company);

    public void delete(Company company);

}
```

PersonDaoImpl

```
public class CompanyDaoImpl
        implements CompanyDao {

    @Override
    public Company get(Long id) {
    ...
    }

    @Override
    public void delete(Company person) {
    ...
    }

 ...
}
```

Konfiguracja DAO

- poprzez konfigurację w XML

```
<bean  id="companyDao"
       class="lab.spring.dao.CompanyDaoImpl">
</bean>
```

- lub poprzez adnotacje

```
@Repository("companyDao")
public class CompanyDaoImpl
       implements CompanyDao{

}
```

Konfiguracja DataSource w Spring

```
<bean id="dataSource"
        class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="jdbcUrl"
                value="jdbc:h2:tcp://localhost/data">
        </property>
        <property name="user" value="sa"></property>
        <property name="password" value=""></property>
        <property name="driverClass"
                value="org.h2.Driver">
        </property>
</bean>

<!-- alternatywnie -->

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

DataSource

- DataSourceUtils
- SmartDataSource
- AbstractDataSource
- SingleConnectionDataSource
- DriverManagerDataSource
- NativeJdbcExtractor

Utworzenie EntityManagerFactory w kontenerze

- pobranie z serwera aplikacyjnego
- LocalEntityManagerFactoryBean
- LocalContainerEntityManagerFactoryBean

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"/>
```

LocalEntityManagerFactoryBean

- Najprostsza 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>
```

LocalContainerEntityManagerFactory Bean

- Najbardziej zaawansowana wersja pozwalająca na zdefiniowanie wszystkich aspektów konfiguracji JPA na poziomie kontenera Spring.
- Nie ma konieczności definiowania pliku persistence.xml

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

Przykład konfiguracji

```
<bean id="myEmf"
        class="org.springframework.orm.jpa.      LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.
                        HibernateJpaVendorAdapter">
        <property name="showSql" value="true" />
        <property name="generateDdl" value="autp" />
        <property name="databasePlatform"
                        value="org.hibernate.dialect.H2Dialect" />
        </bean>
    </property>
   <property name="persistenceUnitName" value="test" />
   <property name="packagesToScan">
      <list>
        <value>com.foo.type</value>
      </list>
   </property>
</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.

web.xml

```
<filter>
   <filter-name>oemv-filter</filter-name>
   <filter-class>
            org.springframework.orm.jpa.support.
                     OpenEntityManagerInViewFilter
   </filter-class>
   <init-param>
      <param-name>
            entityManagerFactoryBeanName
      </param-name>
      <param-value>
        emf
      </param-value>
   </init-param>
</filter>

<filter-mapping>
   <filter-name>oemv-filter</filter-name>
   <servlet-name>spring-servlet</servlet-name>
</filter-mapping>
```

Rozszerzenia

- Grepo
- Spring Data JPA

Spring TX

Mechanizmy transakcji
w środowisku Spring Framework

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.

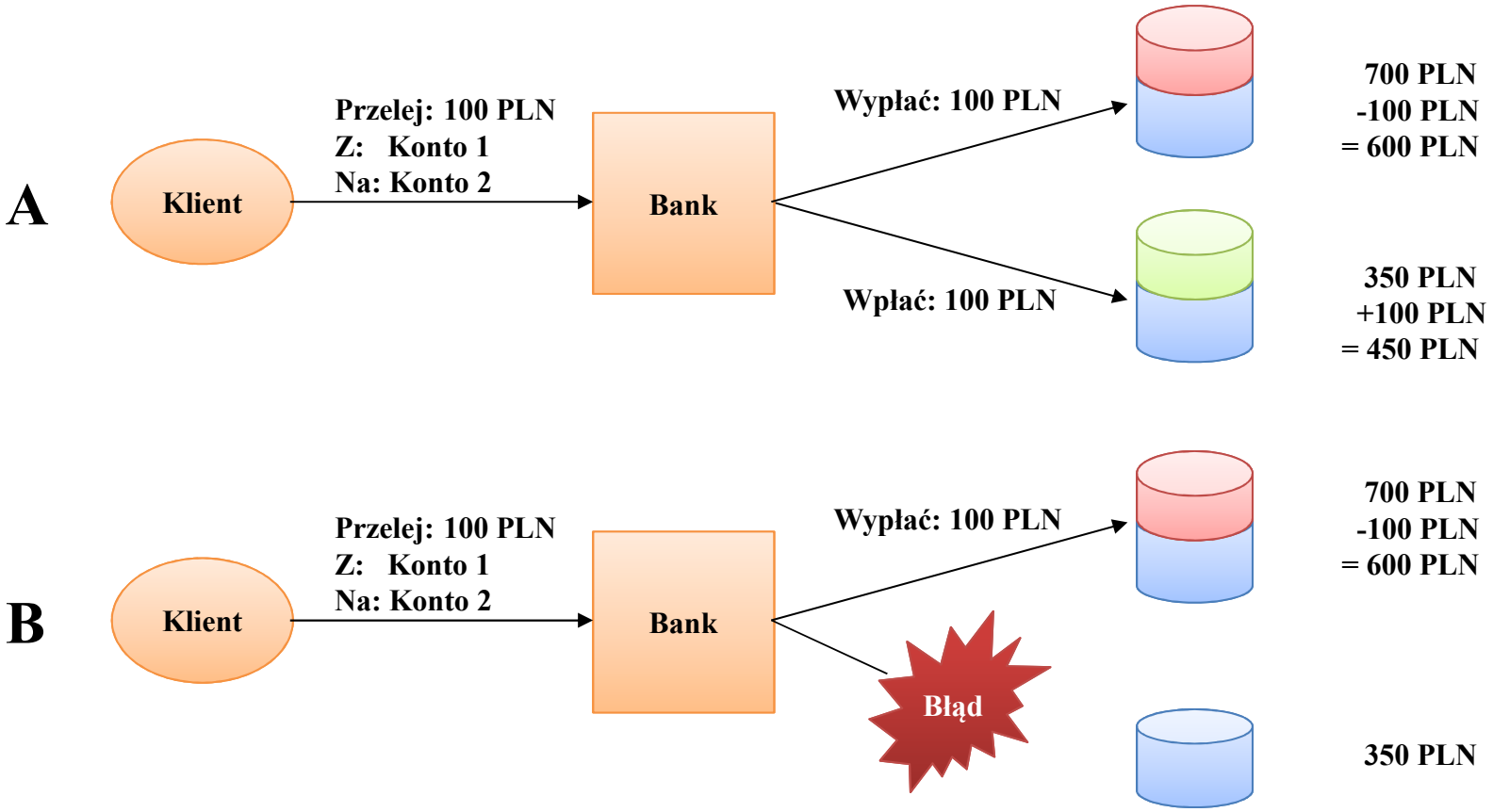
VAVATECH

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)

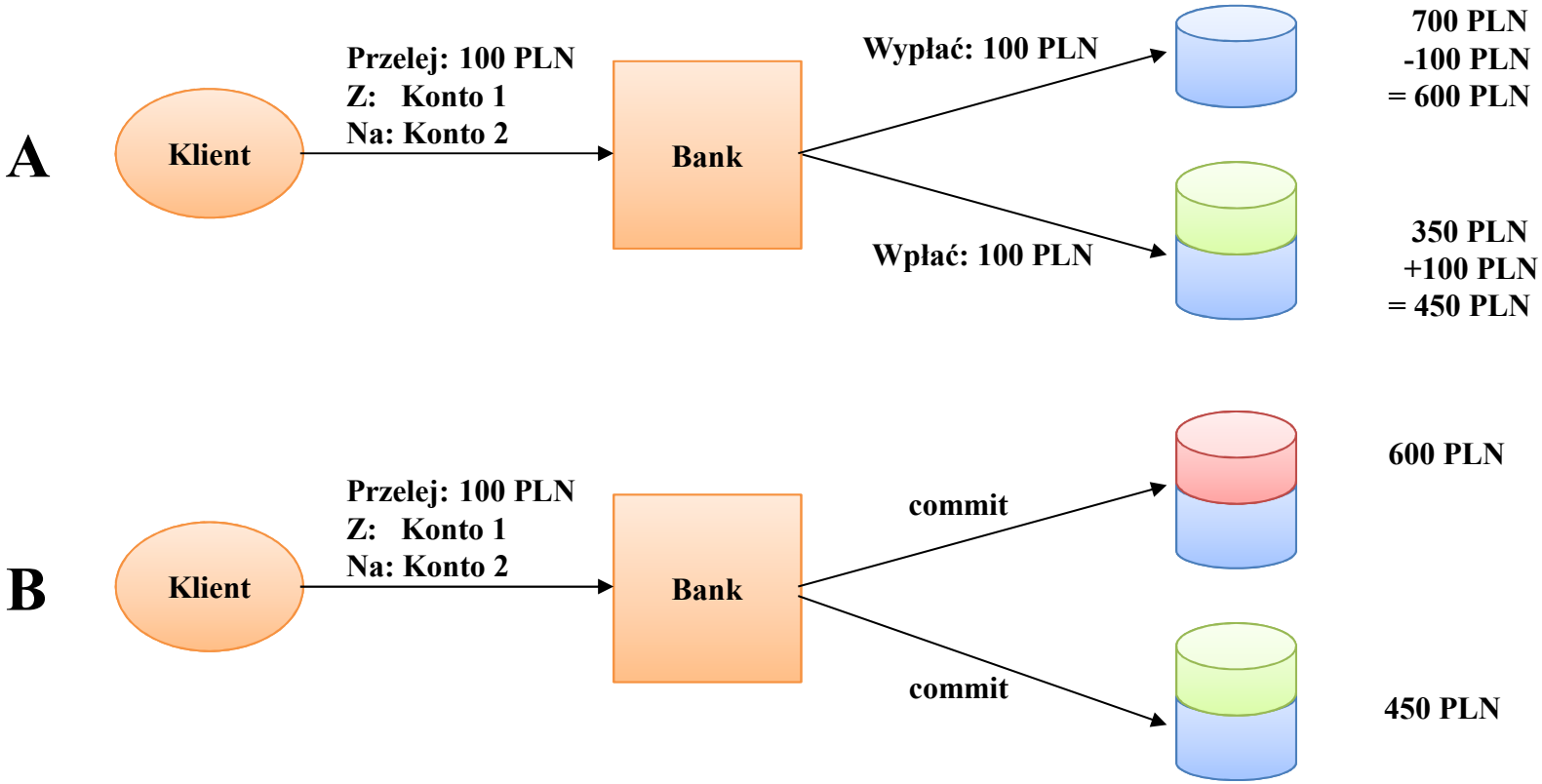
Transakcje lokalne

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



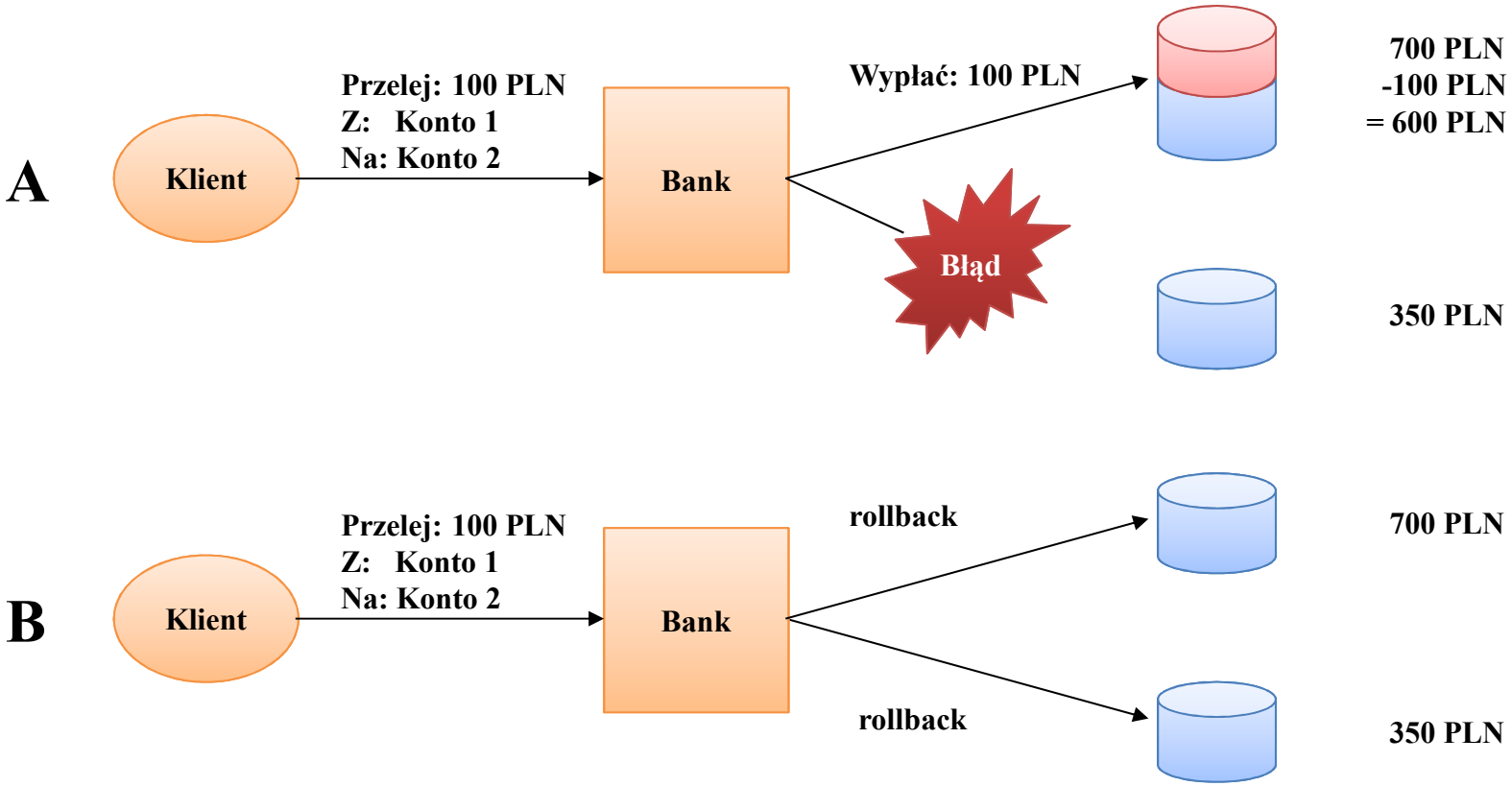
Transakcje lokalne

- 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)



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

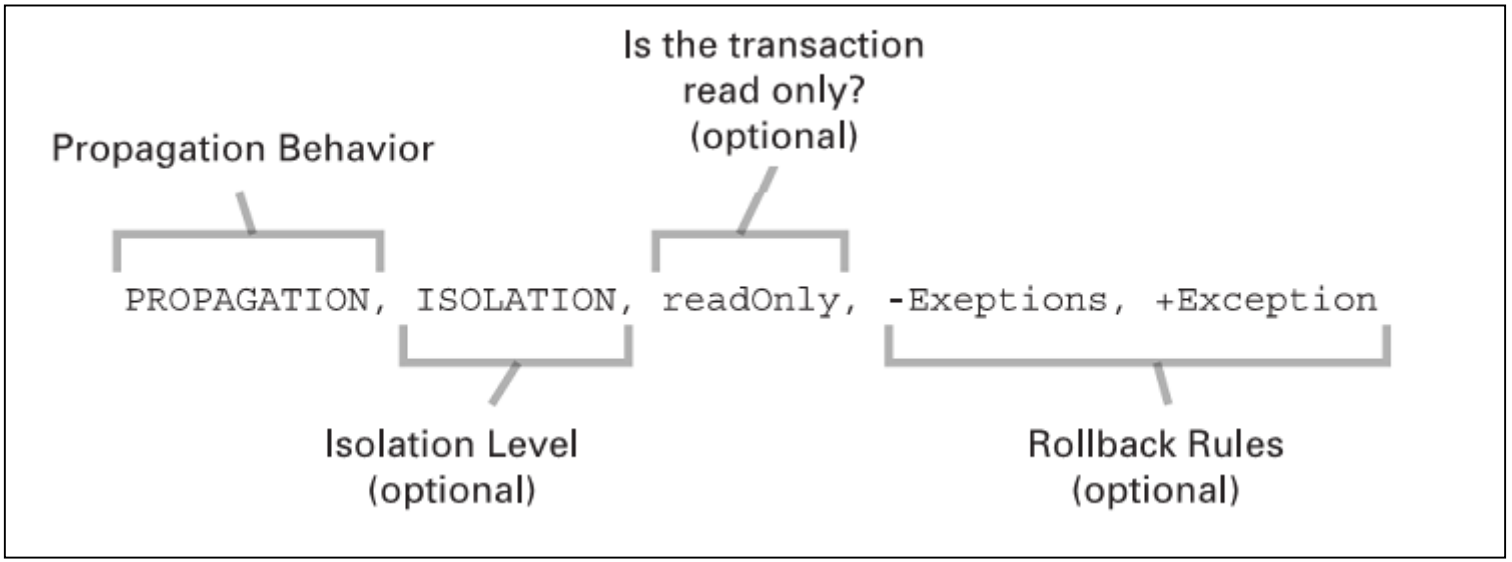
```
<bean id="PersonService"
class="org.springframework.transaction.interceptor.
        TransactionProxyFactoryBean">
    <property name="target">
        <bean class="service.PersonServiceImpl">
        <property name="PersonDao" ref="PersonDao" />
        </bean>
    </property>
    <property name="transactionAttributes">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

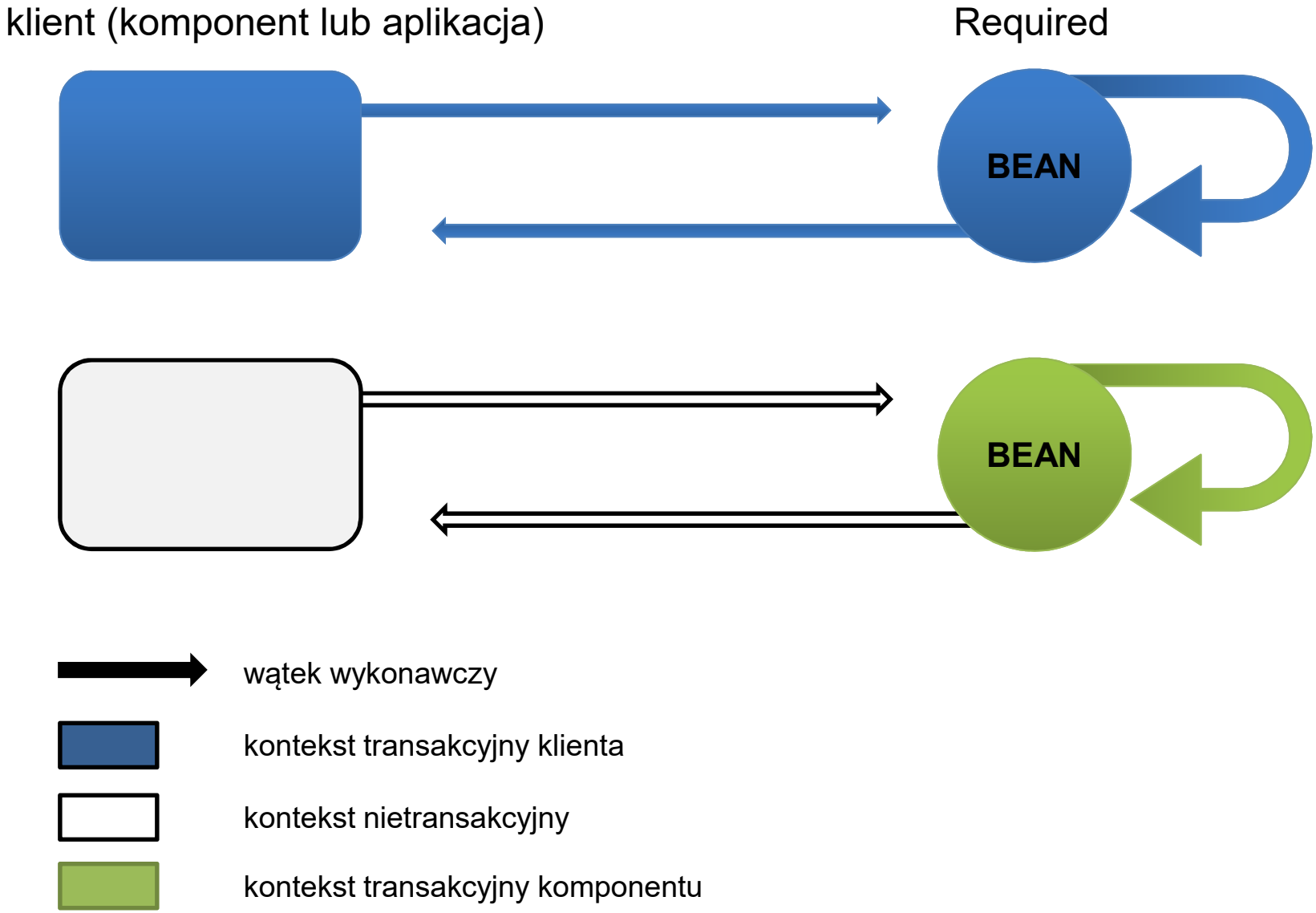
- Propagacja transakcji
- Poziom izolacji
- Atrybuty read only
- Timeout



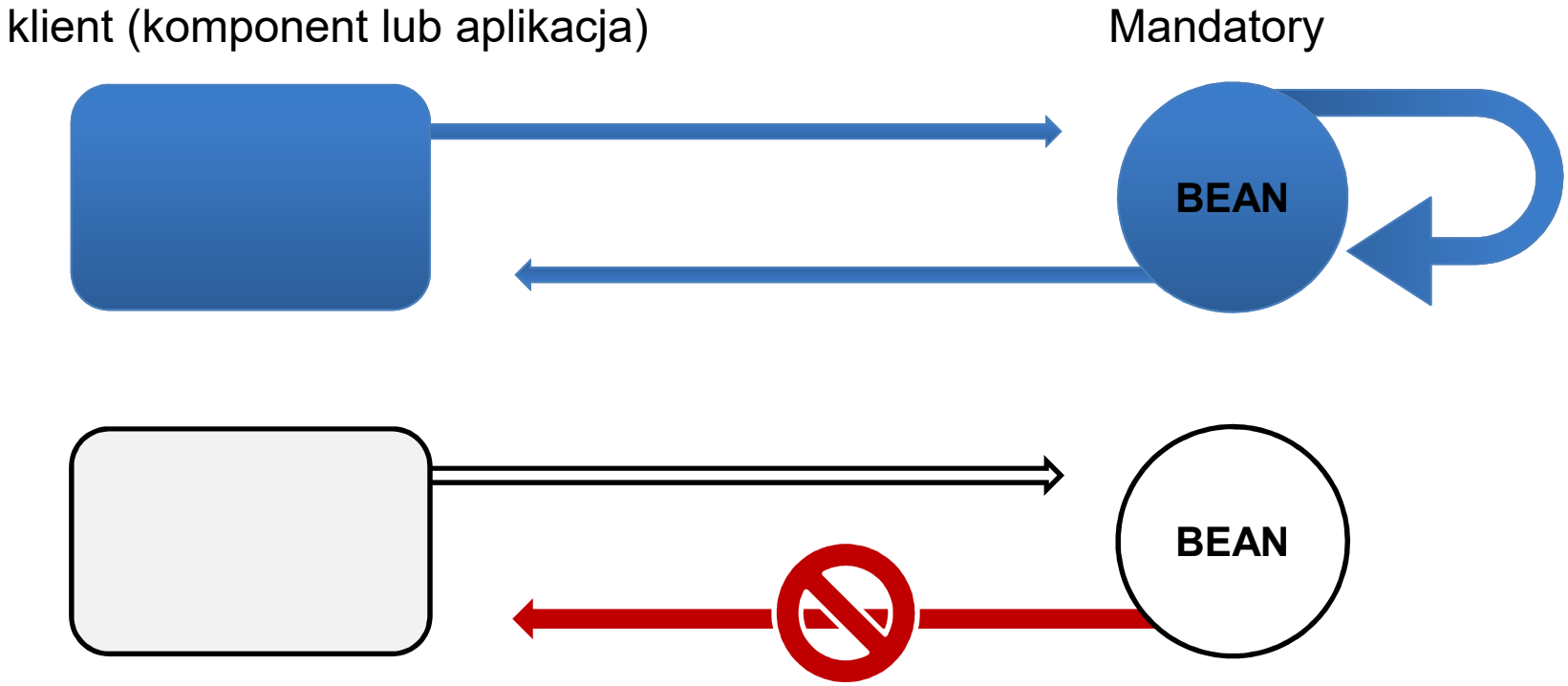
Propagacja transakcji

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

PROPAGATION_REQUIRED



PROPAGATION_MANDATORY



Poziom izolacji transakcji

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

Przykład

T1

```
SELECT * FROM users WHERE
age BETWEEN 10 AND 30;
```

T2

```
INSERT INTO users VALUES
( 3, 'Bob', 27 );
COMMIT;
```

```
SELECT * FROM users WHERE
age BETWEEN 10 AND 30;
```

-Repeatable Read – Phantom Reads)

Przykład

T1

```
SELECT * FROM users WHERE
id = 1;
```

```
SELECT * FROM users WHERE
id = 1; COMMIT;
```

T2

```
UPDATE users SET age = 21
WHERE id = 1;
```

```
ROLLBACK;
```

-Read Uncommited – DIRTY READ

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>
```

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 />
```

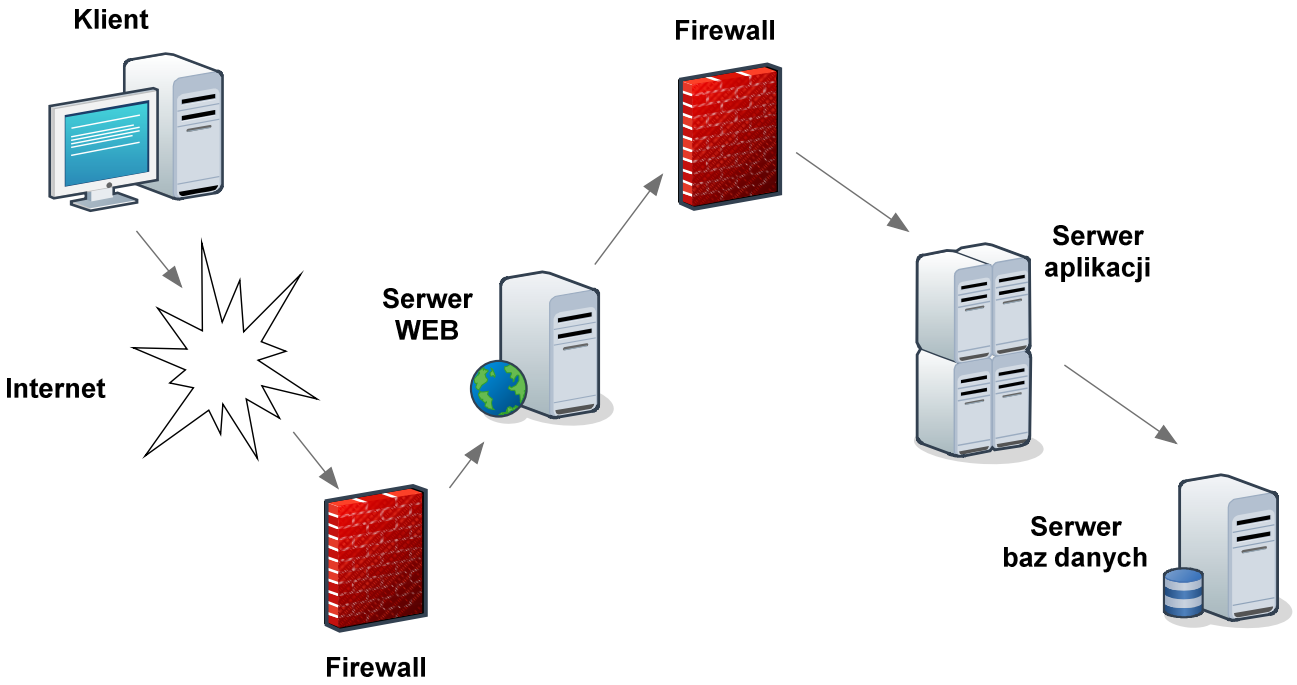
Atrybuty @Transactional

`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

Spring Security

Zabezpieczanie systemu komputerowego

- Problemy bezpieczeństwa powinny być rozpatrywane kompleksowo, na każdym etapie przetwarzania informacji w systemie komputerowym.



Zagadnienia bezpieczeństwa

- **Uwierzytelnianie**
  - weryfikacja tożsamości użytkownika
- **Autoryzacja**
  - weryfikacja praw dostępu użytkownika do określonych zasobów
- **Audyt**
  - zapis zdarzeń związanych z bezpieczeństwem systemu informatycznego
- **Zabezpieczanie kanałów komunikacyjnych**
  - SSL

Historia

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

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.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security.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>
```

Authentication manager

```
<security:authentication-manager alias="manager">
        <security:authentication-provider
          user-service-ref="customUserDetailsService">
         <security:password-encoder ref="passwordEncoder"/>
        </security:authentication-provider>
</security:authentication-manager>
```

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

Przykładowa konfiguracja

```
<security:http auto-config="true" use-expressions="true"
        access-denied-page="/app/auth/denied" >
  <security:intercept-url pattern="/app/auth/login" access="permitAll"/>
  <security:intercept-url pattern="/app/main/admin"
        access="hasRole('ROLE_ADMIN')"/>
  <security:intercept-url pattern="/app/main/common"
        access="hasRole('ROLE_USER')"/>

  <security:form-login
    login-page="/app/auth/login"
    authentication-failure-url="/app/auth/login?error=true"
    default-target-url="/app/main/common"/>

  <security:logout
    invalidate-session="true"
    logout-success-url="/app/auth/login"
    logout-url="/app/auth/logout"/>

 </security:http>
```

Domyślna konfiguracja

```
<security:http>
        <security:form-login />
        <security:http-basic />
        <security:logout />
</security:http>
```

Minimalna konfiguracja

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

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.html'
                  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

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

<http>
  ...
  <session-management
        invalid-session-url="/sessionTimeout.htm"
        session-fixation-protection="[newSession|migrateSession]/>
</http>

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

Zabezpieczanie ziaren

```
<bean id="bankManagerSecurity"
class="org.springframework.security.access.intercept.aopalliance.
                    MethodSecurityInterceptor">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="accessDecisionManager"/>
<property name="afterInvocationManager" ref="afterInvocationManager"/>
<property name="securityMetadataSource">
        <sec:method-security-metadata-source>
        <sec:protect method="com.mycompany.BankManager.delete*"
        access="ROLE_SUPERVISOR"/>
        <sec:protect method="com.mycompany.BankManager.getBalance"
        access="ROLE_TELLER,ROLE_SUPERVISOR"/>
</sec:method-security-metadata-source>
</property>
</bean>
```