



Spring Core

Basepoint
szkolenia@basepoint.pl
+ 48 691-721-682



Marcin Górski
marcin.gorski@basepoint.pl
+48 507-543-982

Agenda

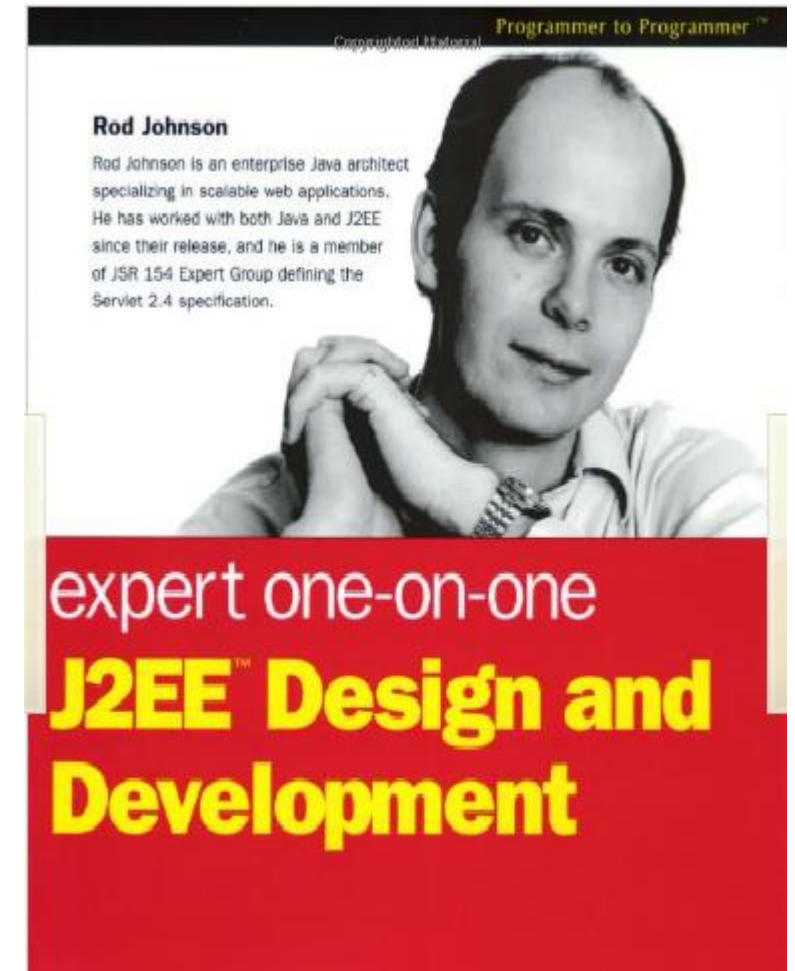
- Przedstawienie trenera oraz uczestników
- Konfiguracja środowiska
- Szkolenie
 - Podstawy, wstrzykiwanie zależności, kontener
 - Konfiguracja (adnotacje, xml)
 - Budowanie aplikacji
 - Aspekty, logowanie, testowanie
 - Persystencja, integracja z Hibernate, transakcje
 - Aplikacje web'owe, REST, Ajax
 - Bezpieczeństwo



Spring - wstęp

Zasada działania springa, architektura.

- 2002
- **Rob Johnson**
Expert One-on-One J2EE Design and Development
- 06.2003 – pierwsza odsłona na licencji Apache 2.0
- M1 udostępniony 03.2004



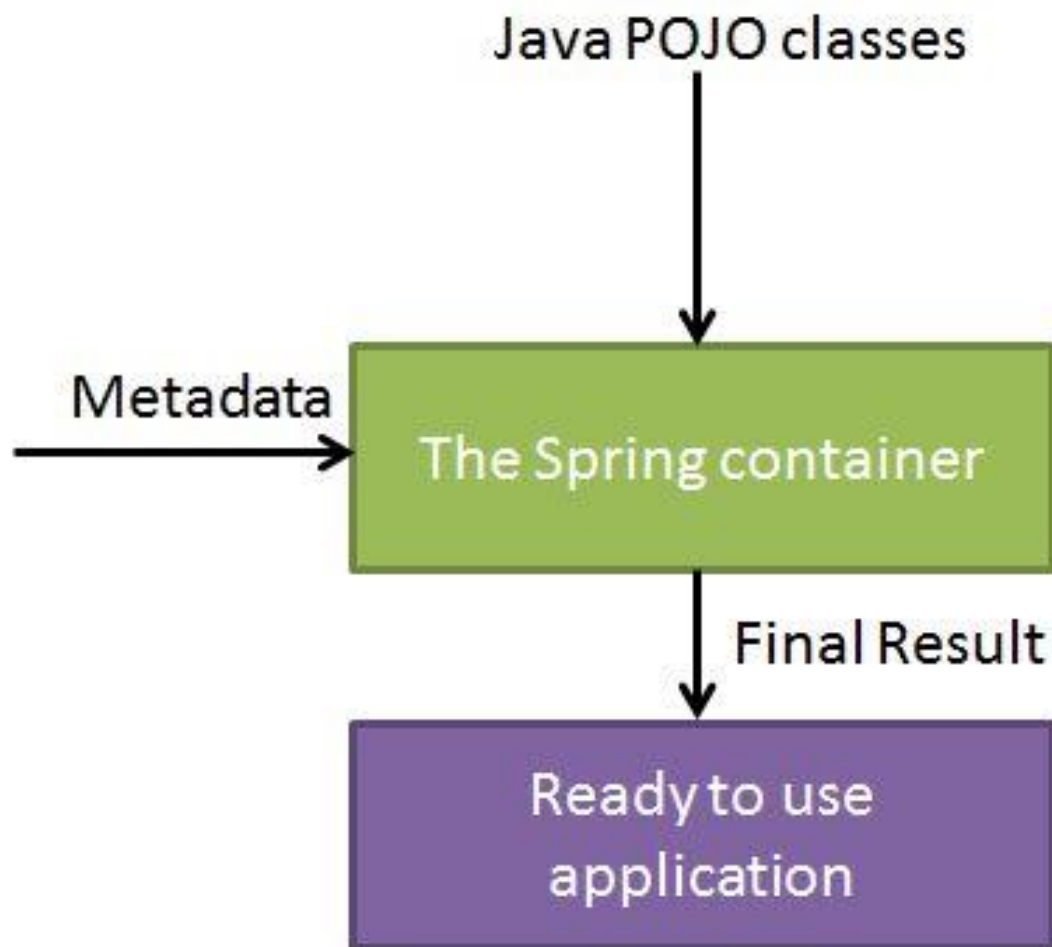
- **Spring 2.0**
Java 1.3AspectJ, JPA
- **Spring 2.5 (11.2007)**
Java 1.4XML namespaces, adnotacje
- **Spring 3.0 (12.2009)**
Java 1.5REST, dodatkowe adnotacje
- **Spring 4.0 (koniec 2013)**
Java SE8, JPA 2.1, Servlet 3.1, event listeners, Java 6



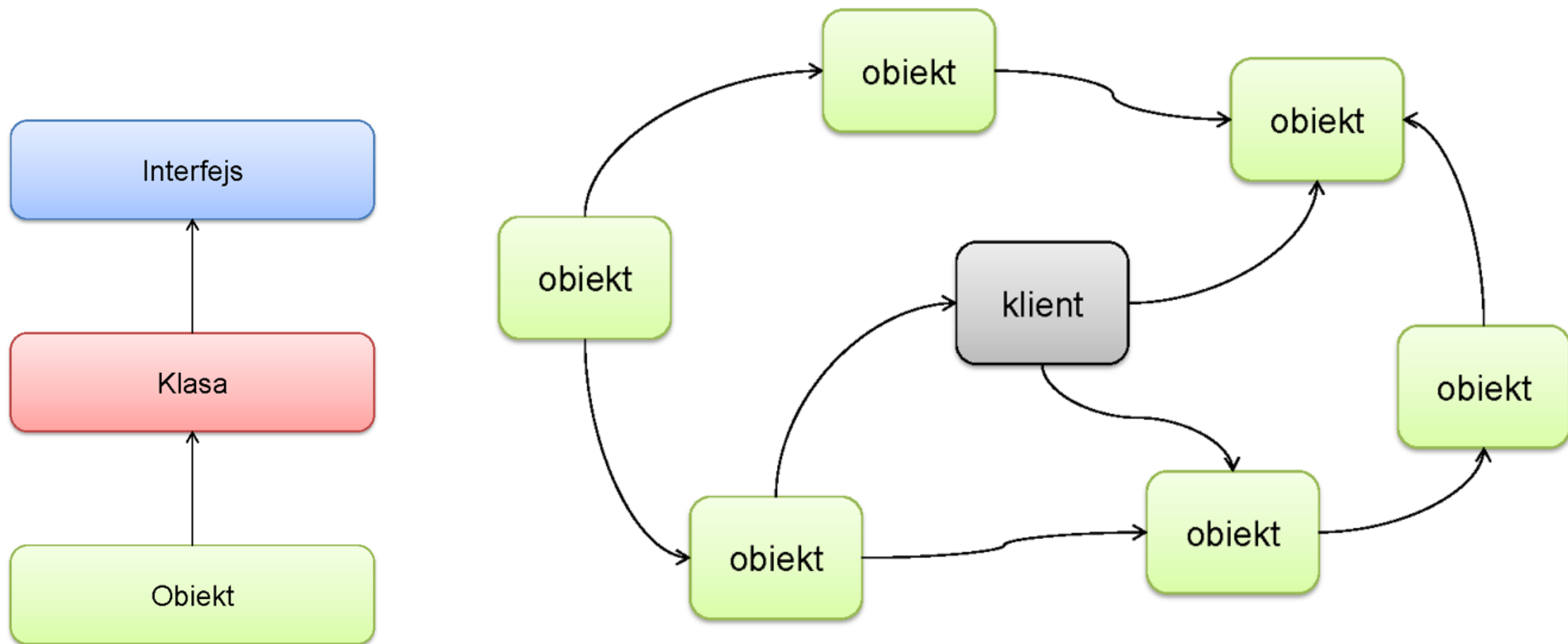
- Lekki framework open-source, Uproszczenie J2EE
- Oparty o POJO
- Modularny
- Rozszerzalny
- Stał się popularnym standardem w Java Enterprise
- Łatwy w integracji z projektem
- Przyspieszenie budowy aplikacji oraz przyspieszenie najczęstszych *use case*
- Nacisk na *good programming conventions* (struktura projektu, nazwy, itp.)
- Programista koncentruje się na logice biznesowej



Budowanie aplikacji w oparciu o Spring



Klasy i obiekty w Javie



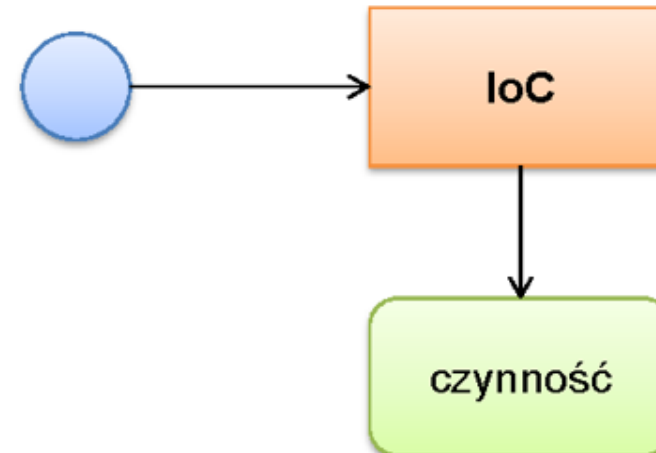
Inversion of Control

- wzorzec architektoniczny polegający na przeniesieniu na zewnątrz komponentu (np. obiektu klasy) odpowiedzialności za kontrolę wybranych czynności
- tzw. Hollywood principle – don't call us, we will call you

PODEJŚCIE TRADYCYJNE



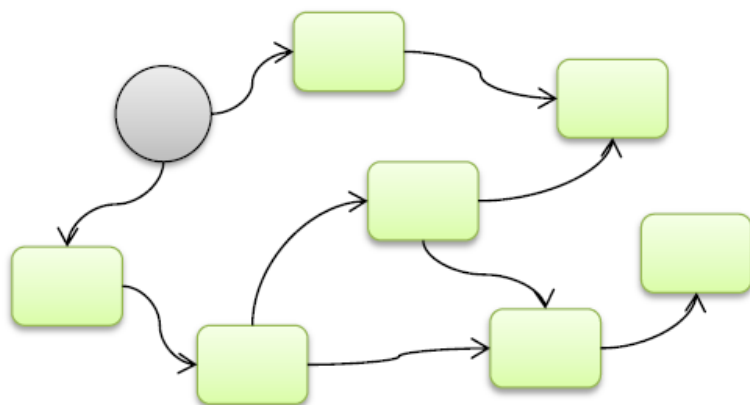
PODEJŚCIE IoC



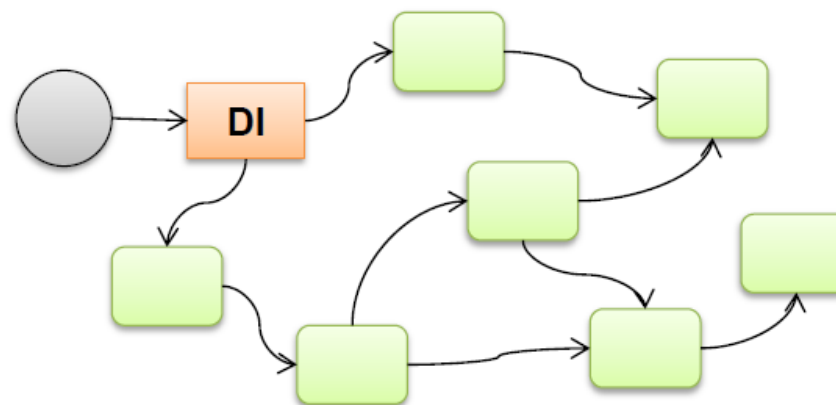
Dependency Injection

- 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

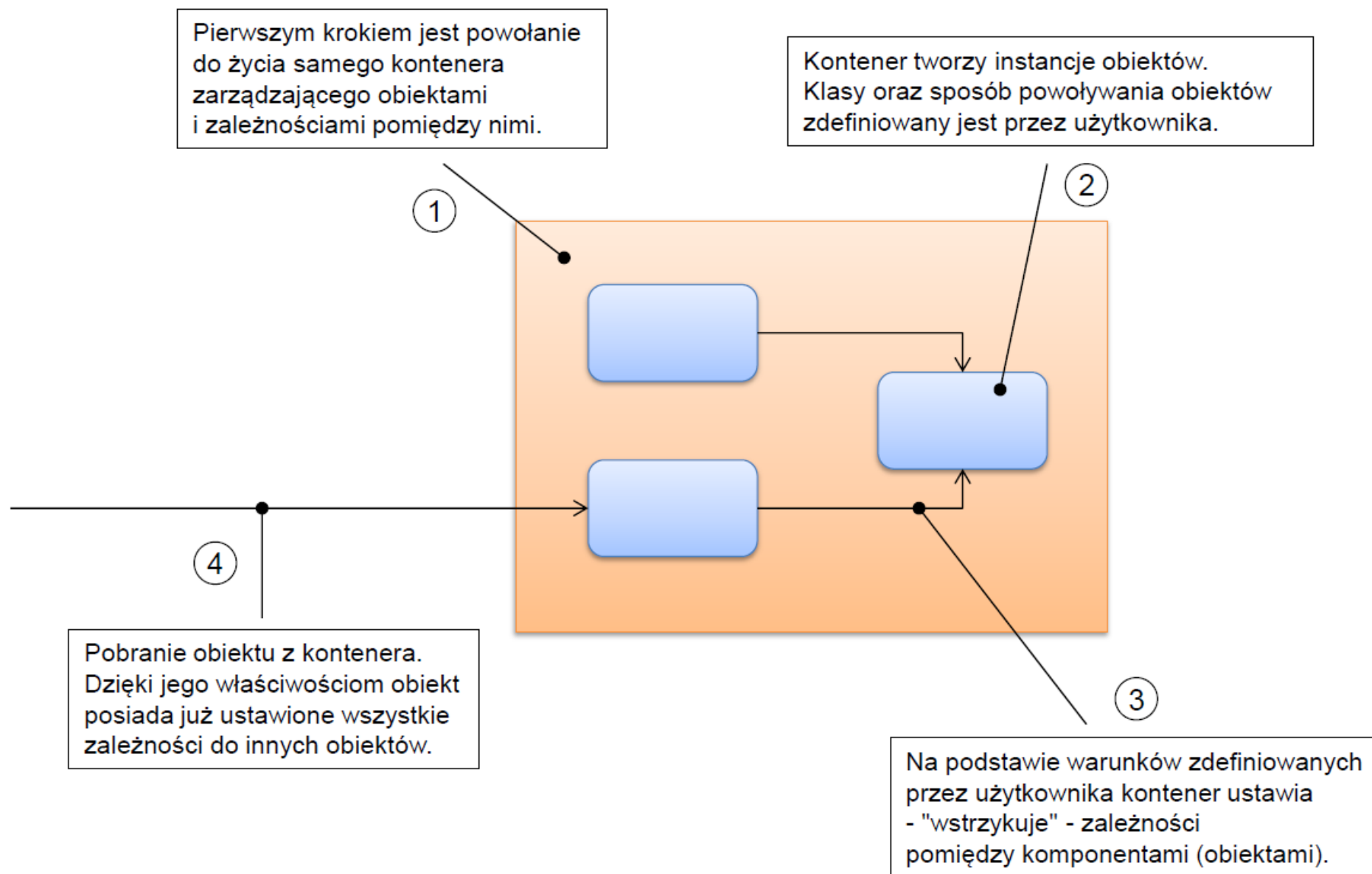
PODEJŚCIE TRADYCYJNE



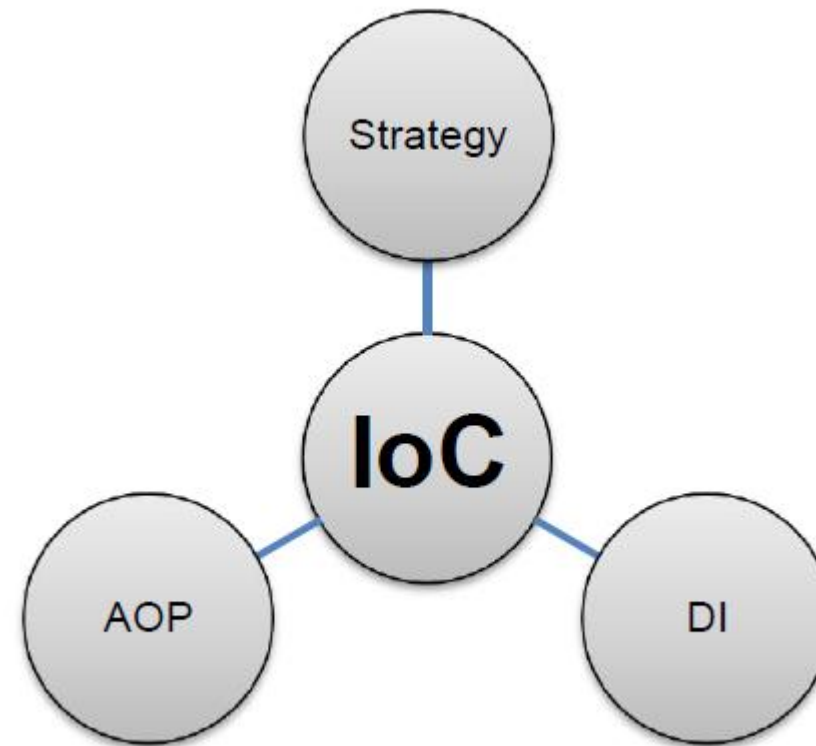
PODEJŚCIE DI



Działanie kontenera DI



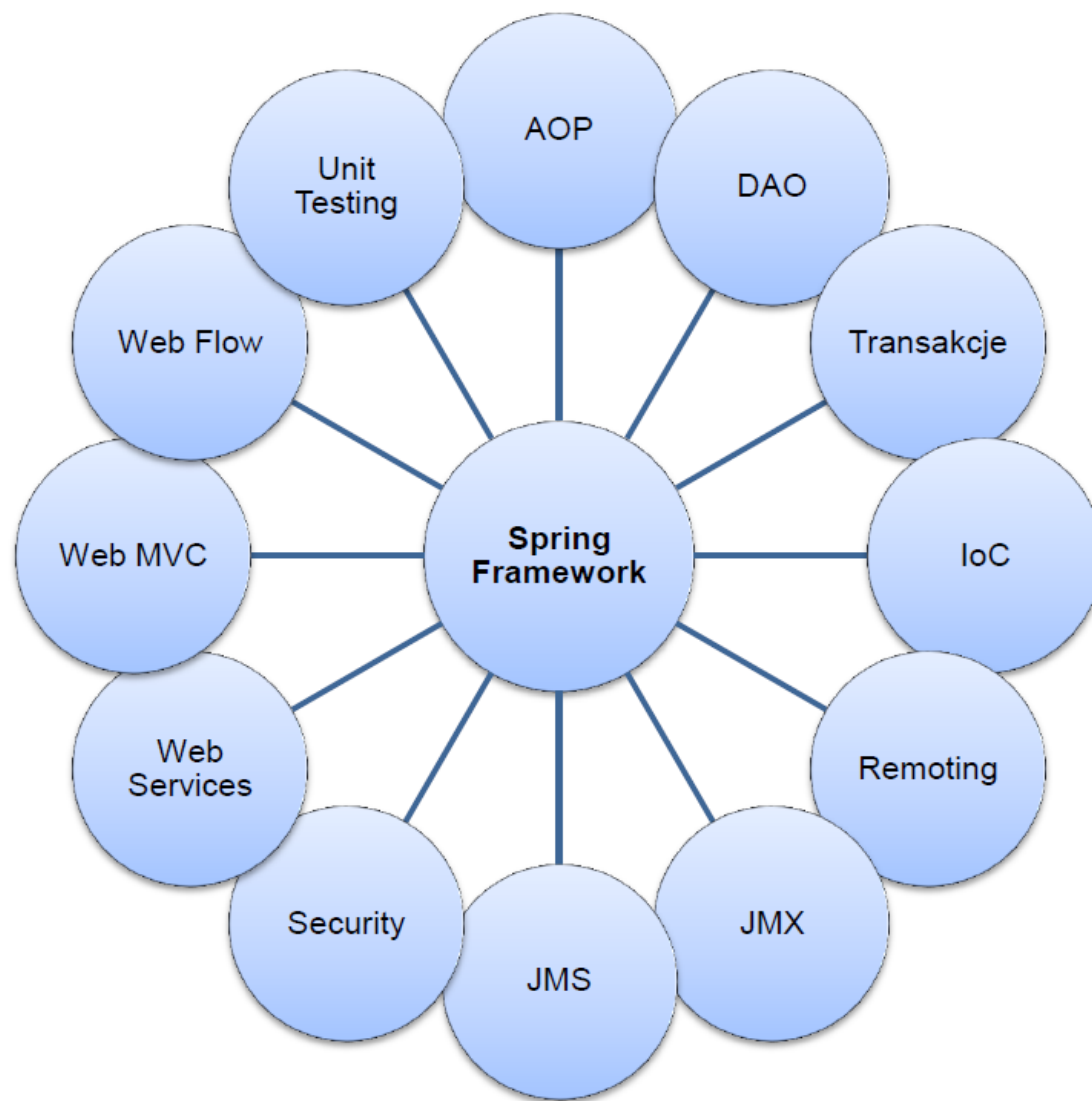
- Inversion of Control często błędnie utożsamiane jest jednoznacznie z Dependency Injection.
- Dependency Injection to szczególny przypadek IoC, który obejmuje szerszy krąg przypadków.



Architektura Spring

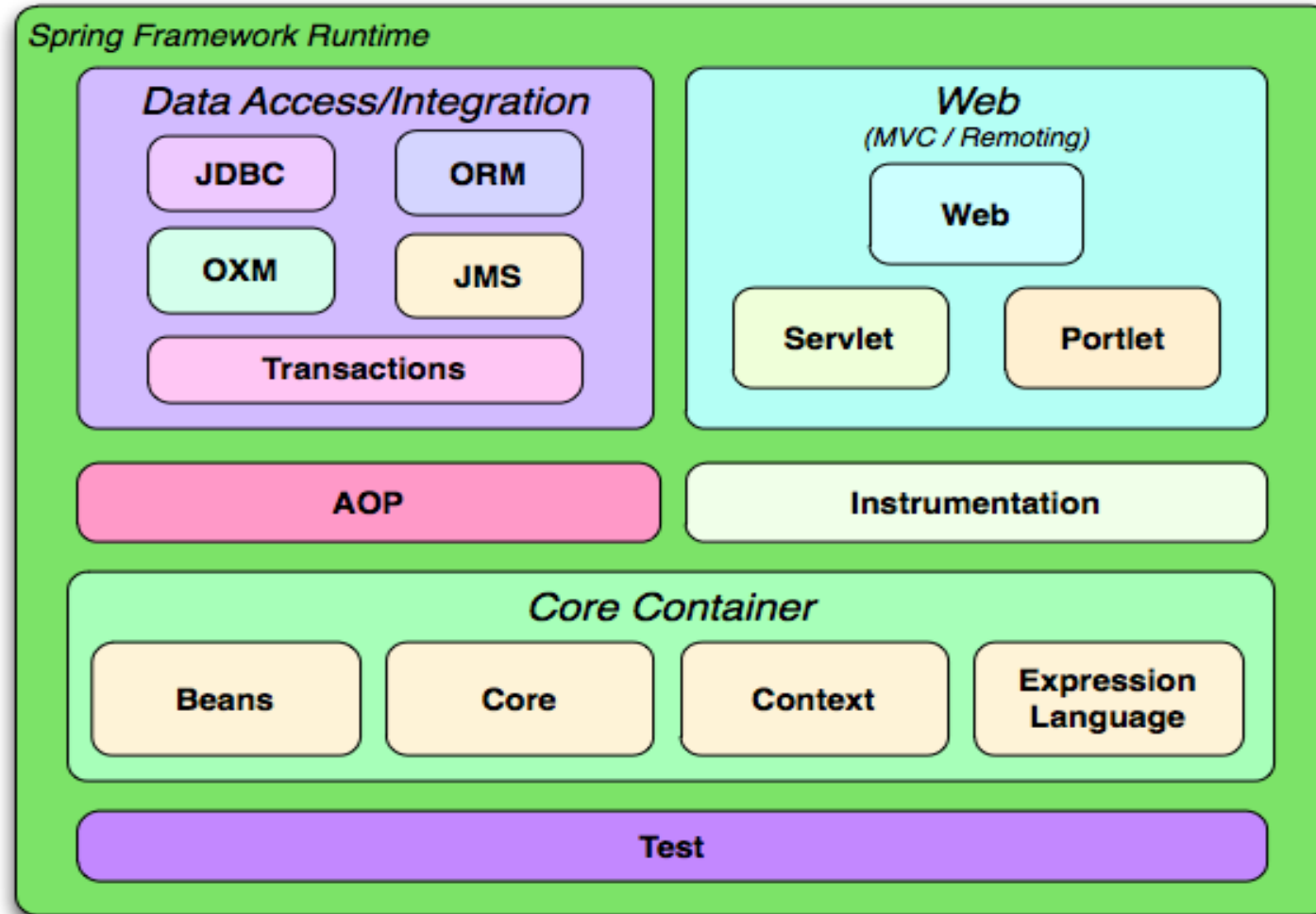
Inversion of Control. Dependency Injection.

Spring framework – budowa modułowa

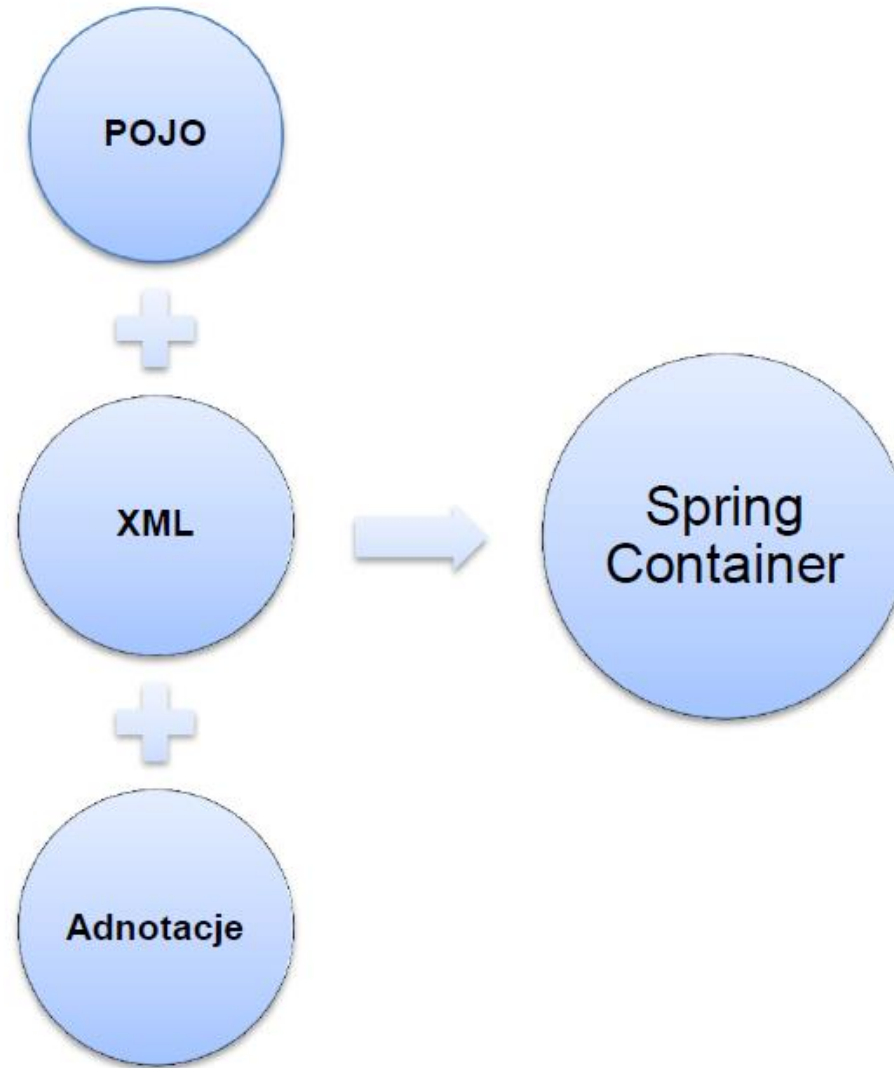


Modułowa budowa

- Framework/IO
- Spring Boot
- MVC
- XD
- Cloud
- Data
- Integration
- Batch
- Roo
- Security
- Social
- Android
- Web Flow



Spring jako kontener



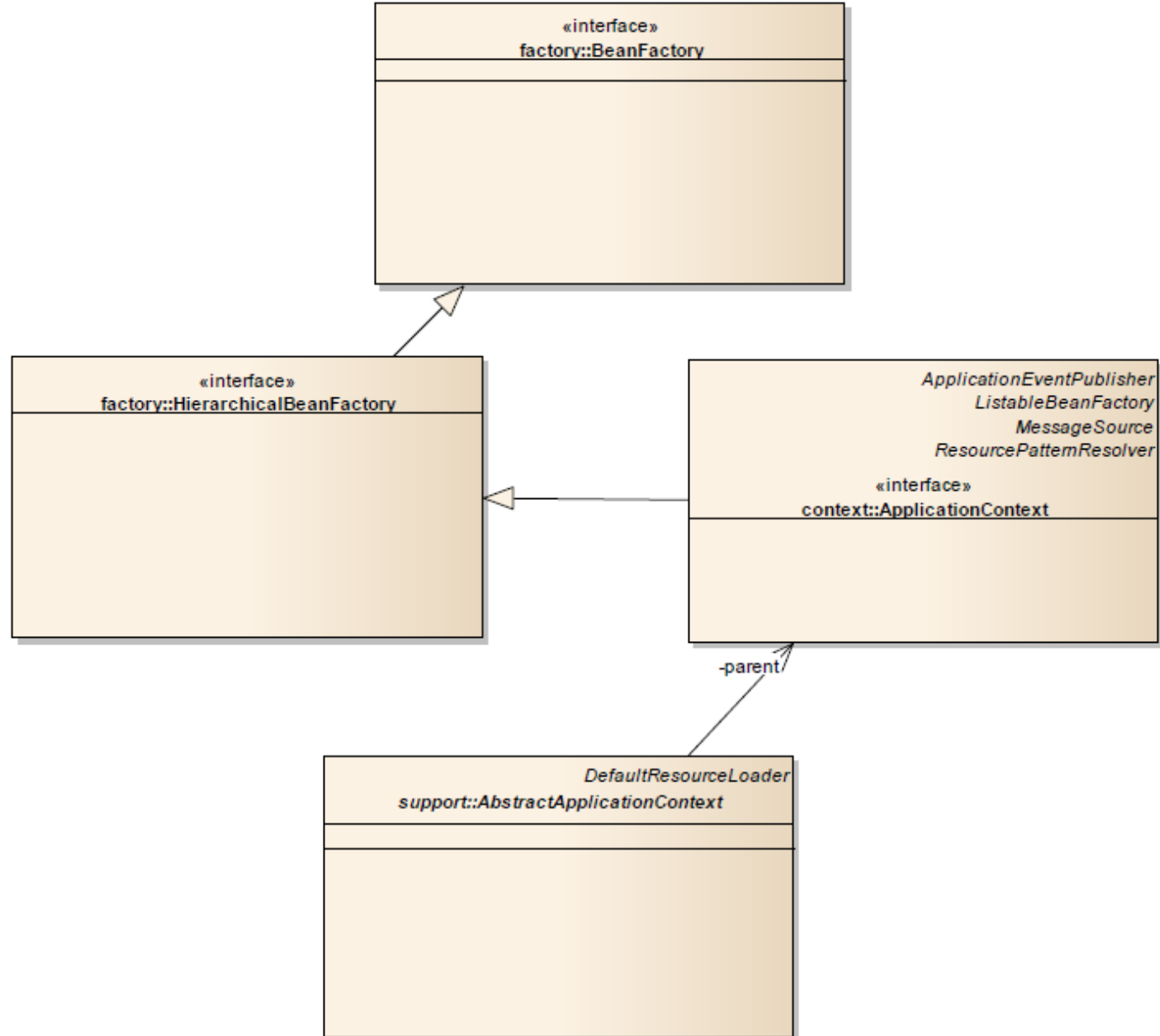
- XML
 - bardziej elastyczny
 - duże pliki z konfiguracją XML
 - odseparowany od kodu Javy (zmiany nie wymagają rekompilacji)
- Adnotacje
 - silne typowanie (pozwala na unikanie błędów)
 - przyjemniejsze kodowanie – większość programistów nie lubi XMLa
- Groovy (nowość Spring 4)
 - zalety XML + zwięzła notacja
- Rozwiązanie hybrydowe



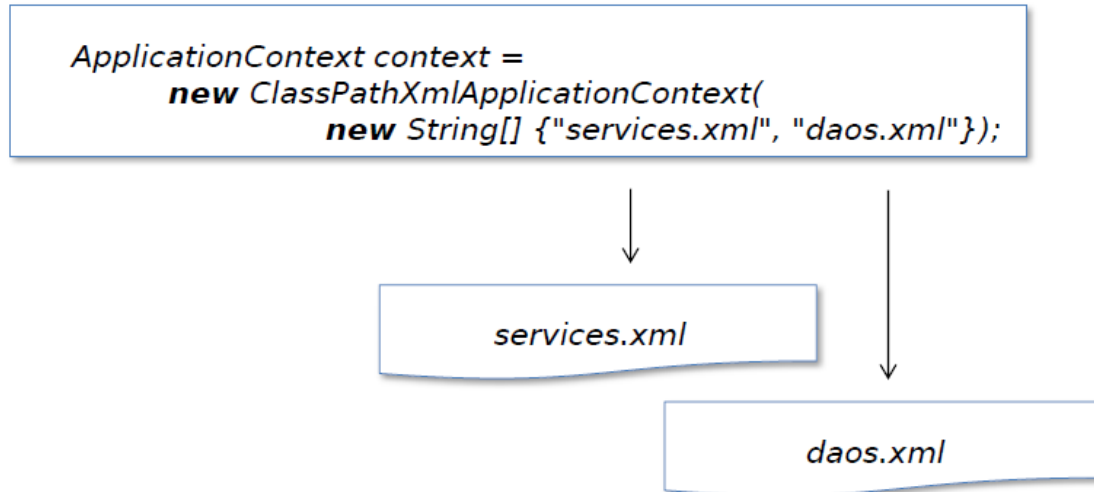
- **Core, Beans**
 - podstawy framework'u, IoC, Dependency Injection
- **Context**
 - Java Naming and Directory Interface (JNDI)
- **Expression language**
 - Dostęp do grafu obiektów w runtime
- Konstrukcja springowego IoC osadzona jest w dwóch pakietach
 - `org.springframework.beans`
 - `org.springframework.context`



Podstawowe interfejsy



Inicjalizacja kontenera



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).

Konfiguracja: 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="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <!-- more bean definitions go here -->

</beans>
```



Definicja bean'a – adnotacje Javowe

- Adnotacje klas
 - @Component
 - @Service
 - @Controller
 - @Repository
 - @RestController
- Adnotacje metod
 - @Bean

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = { "com.trainings.spring.basic" })



Definicje bean'a – Groovy (nowość Spring 4.0)

```
def reader = new GroovyBeanDefinitionReader(myApplicationContext)
reader.beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
        settings = [mynew:"setting"]
    }
    sessionFactory(SessionFactory) {
        dataSource = dataSource
    }
    myService(MyService) {
        nestedBean = { AnotherBean bean ->
            dataSource = dataSource
        }
    }
}
```



Rozszerzanie konfiguracji opartej o 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.



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>*



Upraszczenie konfiguracji – namespace util

```
<?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="emails"
class="org.springframework.beans.factory.config.ListFactor
yBean">
<property name="sourceList">
<list>
<value>one@gmail.com</value>
<value>two@gmail.com</value>
</list>
</property>
</bean>

</beans>
```

```
<?xml version="1.0" encoding="UTF-8"?><beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:util="http://www.springframework.org/schema/ut
il" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://www.springframework.org/s
chema/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">

<util:list id="emails" list-
class="java.util.ArrayList">
<value>one@gmail.com</value>
<value>two@gmail.com</value>
</util:list>

</beans>
```



Złożony plik konfiguracyjny

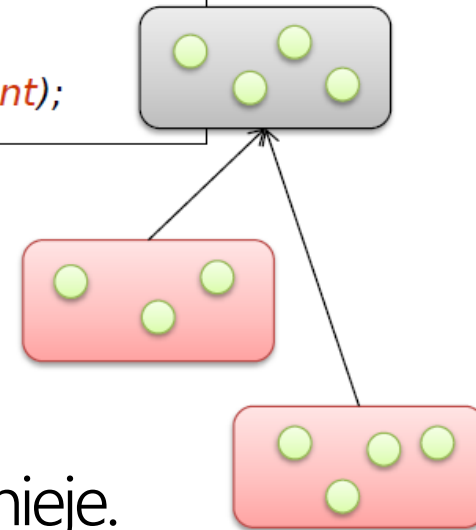
- Złożony plik konfiguracyjny daje szerokie możliwości konfiguracji systemu.
- Pozwala utrzymać większy porządek dzięki podziałowi na mniejsze części, umożliwia wprowadzenie dynamicznego zestawu komponentów w zależności od zawartych np. w classpath bibliotek

```
<?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>
```

Kilka kontenerów

```
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.



Spring Context

Definicja komponentów

- **Inicjalizacja**
 - tworzenie serwisów
 - alokacja zasobów
- **Użycie**
 - 99% czasu
- **Destrukcja**
 - zwolnienie zasobów
 - niszczenie obiektów



Tworzenie instancji bean'a

- Poprzez konstruktor
- Factory method
- Z użyciem instancji innego bean'a
- FactoryBean
- Statyczną metodą



Najprostszy bean

- XML

```
<bean id="myBean"  
class="com.trainings.MyBean"/>
```

- Wsparcie dla adnotacji

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

- Adnotacje

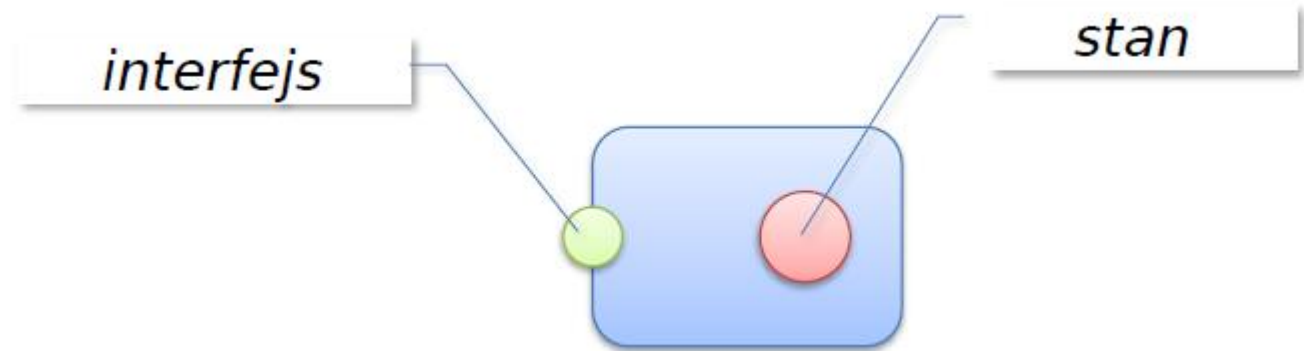
```
@Configuration  
@ComponentScan( basePackages = "com.trainings.beans")  
public class AppConfig {  
    @Bean(name = "myBean")  
    public MyBean createBeanWd() {  
        return new MyBean();  
    }  
}
```

```
@Component  
public class MyBean {}
```



Definicja komponentów


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



Definicja bean'a

- Elementy tagu `<bean />`
 - class
 - name
 - scope
 - constructor arguments
 - properties
 - autowiring mode
 - lazy-initialization mode
 - initialization method
 - destruction method



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

@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.
- Domyślna nazwa na podstawie nazwy klasy:
 - `com.trainings.MyBean` → `myBean`



- 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>
```

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

```
<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.



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



```
<bean id="myBean" class="com.examples.MyBean"/>
```



Właściwości klasy na przykładzie DataSource

```
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) {
    ...
    }
}
```



Ustawianie właściwości – poprzez wywołanie settera !

```
<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>${jdbc.username}</value> </property>  
  <property name="password"> <value>${jdbc.password}</value> </property>  
  
</bean>
```

Konfiguracja bean'a – konstruktor vs settery

- **Konstruktor**

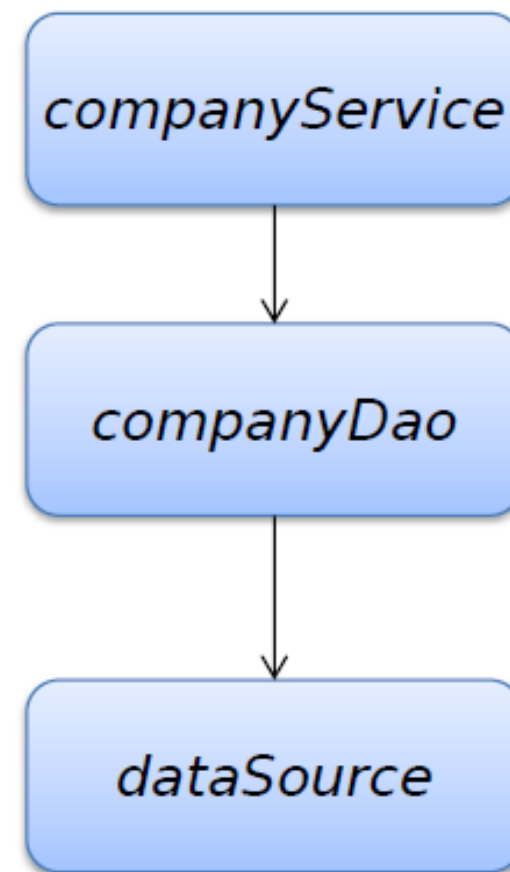
- wymagane zależności
- niezmiennie wartości (immutable)

- **Settery**

- zależności opcjonalne – oparte o wartości domyślne

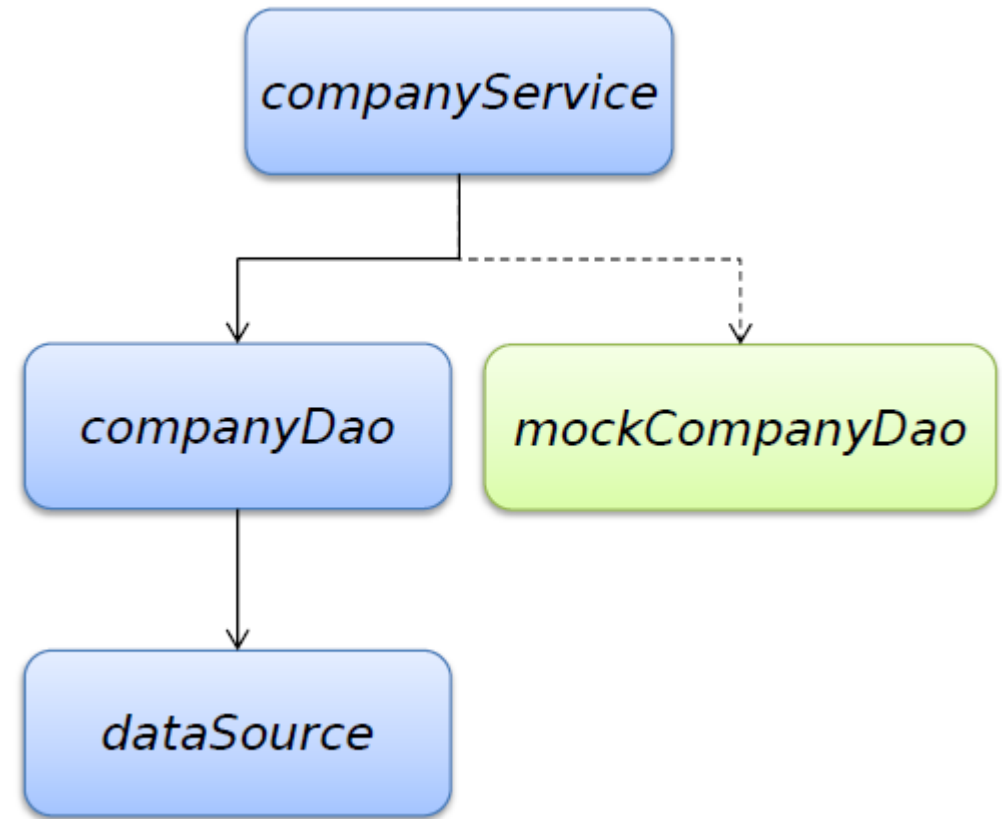
Zależności pomiędzy obiektami

- 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" – warstwy sprzyja przejrzystości oraz ułatwia utrzymanie
- Często spotykany podział MSVC
model <> service <> controller <> view



Interface'y vs 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 – zależności między obiektami

```
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;  
    }  
  
}
```

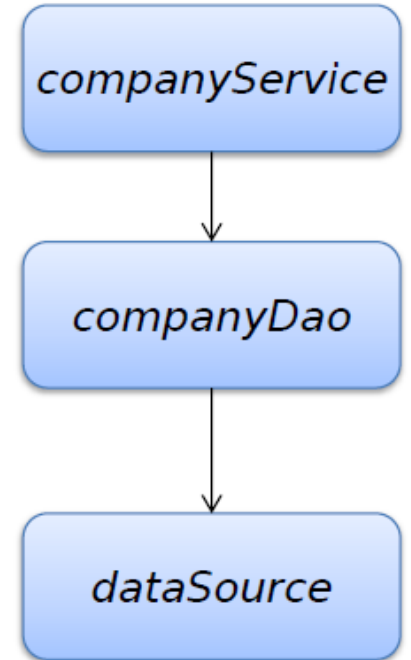


Zależności między obiektami w Spring

```
<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 – kolejne uproszcze zapisu XML

```
<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">

<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>
</beans>
```



Namespace p - referencja

```
<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>
```

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

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



Wstrzykiwanie zależności - 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;  
    }  
  
}
```



Konstruktor z parametrami

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <!-- z zachowaniem kolejności -->  
    <constructor-arg value="1"/>  
    <constructor-arg value="2"/>  
</bean>
```

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

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



Metoda fabrykująca obiektu (Factory method)

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

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

```
<bean id="anotherBean" class="example.AnotherBean"  
    factory-method="createInstance" />
```

brak atrybutu class !



metoda statyczna w klasie



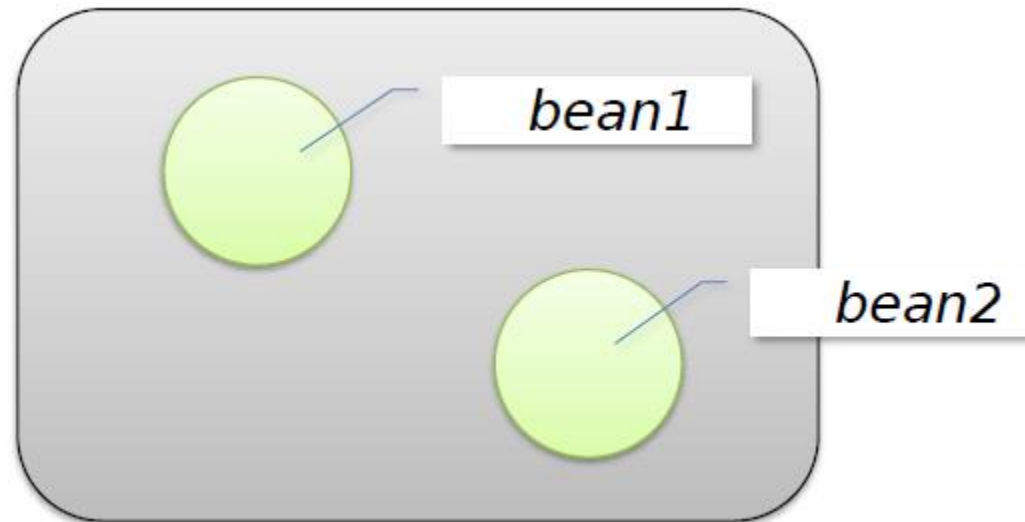
Implementacija interfejsu Factory Bean

```
public interface FactoryBean<T> {  
    T getObject() throws Exception;  
    Class<?> getObjectType();  
    boolean isSingleton();  
}
```



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);
```



Spring Beans

Realizacja IoC za pomocą Spring



Sposoby wiązania bean'ów (autowire)

- Istnieje możliwość zdefiniowania domyślnego automatycznego wiązania i jego rodzaju poprzez dodanie atrybutu do elementu *beans* w pliku konfiguracyjnym.
- Możliwe ustawienia **default-autowire**
 - no (domyślne)
 - byName
 - byType
 - constructor
 - autodetect



Autowire

```
@Component
public class CompanyServiceImpl implements CompanyService {

    @Autowired
    private CompanyDao companyDao;

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

```
@Component
public class CompanyDaoImpl implements CompanyDao{

    @Autowired
    private DataSource dataSource;

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



Autowire

```
@Service
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{ }
```

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



Własny @Qualifier

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Qualifier
```

```
public @interface MyCompanyDao {  
}
```

```
@Service
```

```
public class CompanyServiceImpl implements CompanyService {
```

```
@Autowired @MyCompanyDao private CompanyDao companyDao;
```

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

```
    } }
```

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



- JSR-250

`@Resource(name = "myDao")`

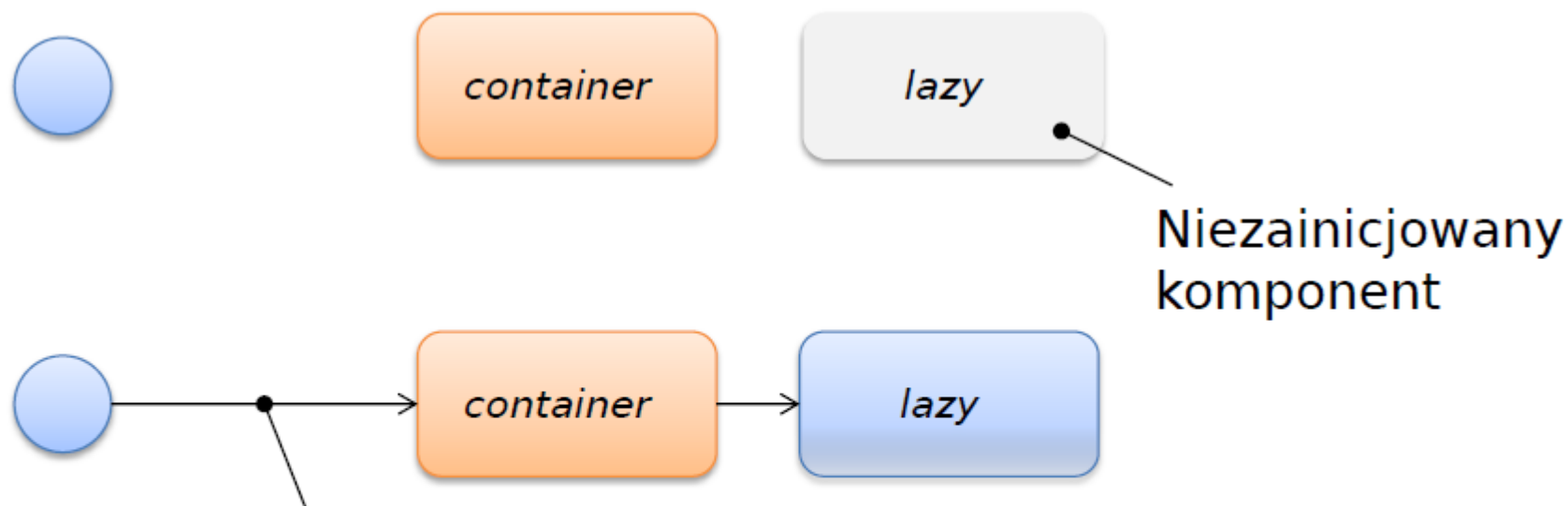
- JSR-330 - javax.annotation

`@Inject @Qualifier("myDao")`



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.

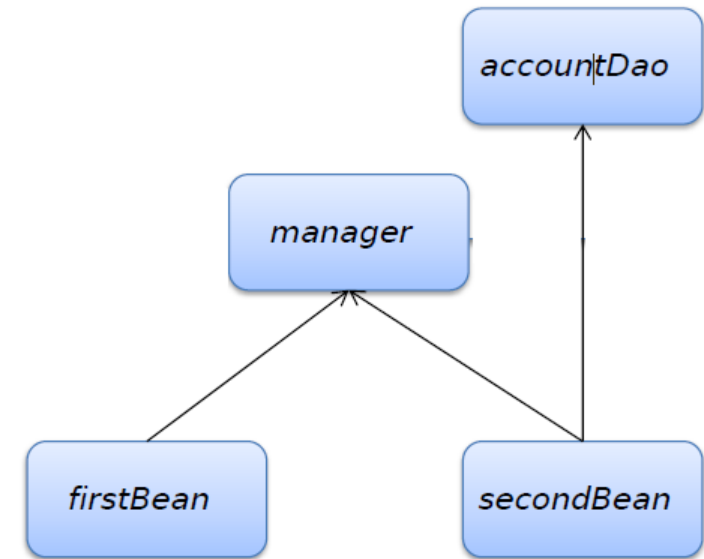
Zależności między bean'ami - depends-on

- Wymusza kolejność inicjalizacji bean'ów

```
<bean id="beanOne" class="ExampleBean"  
      depends-on="costService, accountDao">  
  <property name="deviceManager" ref="deviceManager" />  
</bean>
```

```
<bean id="costService" class="example.service.CostService" />
```

```
<bean id="accountDao" class="example.jdbc.JdbcAccountDao" />
```



Lazy i depends on w adnotacjach

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



Dziedziczenie definicji bean'a

```
<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> 67
```



- `<bean id="parent" abstract="true" class="..."/>`
- `<bean id="service" parent="parent" class=".." />`



- Map
- List
- Set
- Properties



java.util.Properties

```
<property name="emails">  
  <props>  
    <prop key="admin">admin</prop>  
  </props>  
</property>
```

java.util.List

```
<property name="emails">  
  <list>  
    <value>info</value>  
    <ref bean="emailDataSource" />  
  </list>  
</property>
```



java.util.Map

```
<property name="emails">
  <map>
    <entry key="admin" value="..." />
    <entry key="developer" value-ref="..." />
  </map>
</property>
```

java.util.Set

```
<property name="emails">
  <set>
    <value>info</value>
    <ref bean="emailDataSource" />
  </set>
</property>
```



```
<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>
```



Tworzenie bean'a – wartości null vs puste pola

```
<bean class="Contact" >  
    <property name="nickname" value="" />  
</bean>
```

```
<bean class="Contact" >  
    <property name="nickname"><null /></property>  
</bean>
```

- 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 {  
  
    }  
  
}
```



Destrukcja komponentów

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

```
public class PersonServiceImpl implements PersonService, DisposableBean {  
  
    public void destroy() throws Exception {  
  
    }  
  
}
```



Utworzenie i destrukcja - adnotacje

@PostConstruct

```
public void init(){  
}
```

@PreDestroy

```
public void destroy(){  
}
```



Zasięg bean'a (scope)

- Możliwe ustawienia
 - singleton (domyślnie)
 - prototype
 - session
 - request
 - globalScope
 - custom



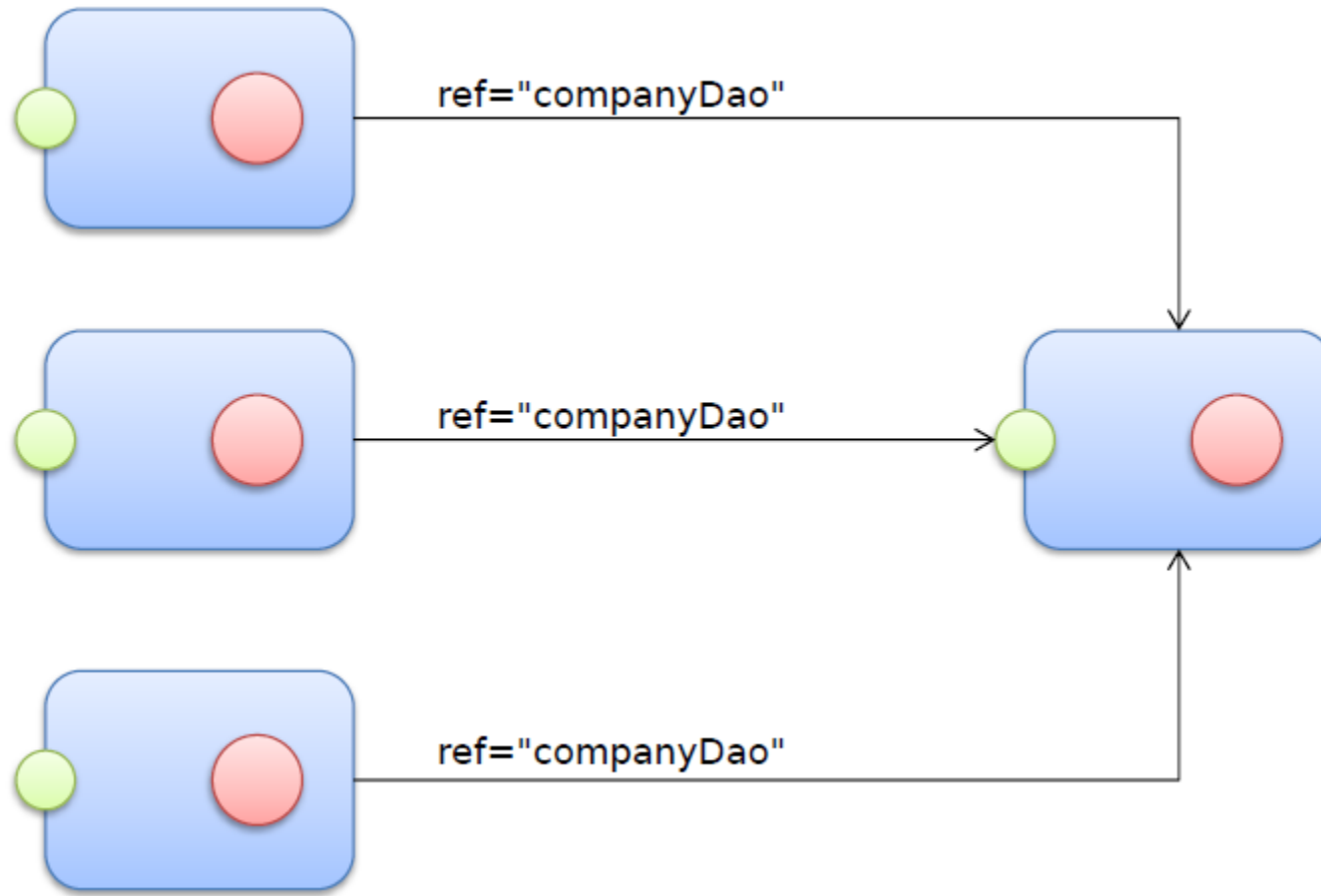
Scope

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

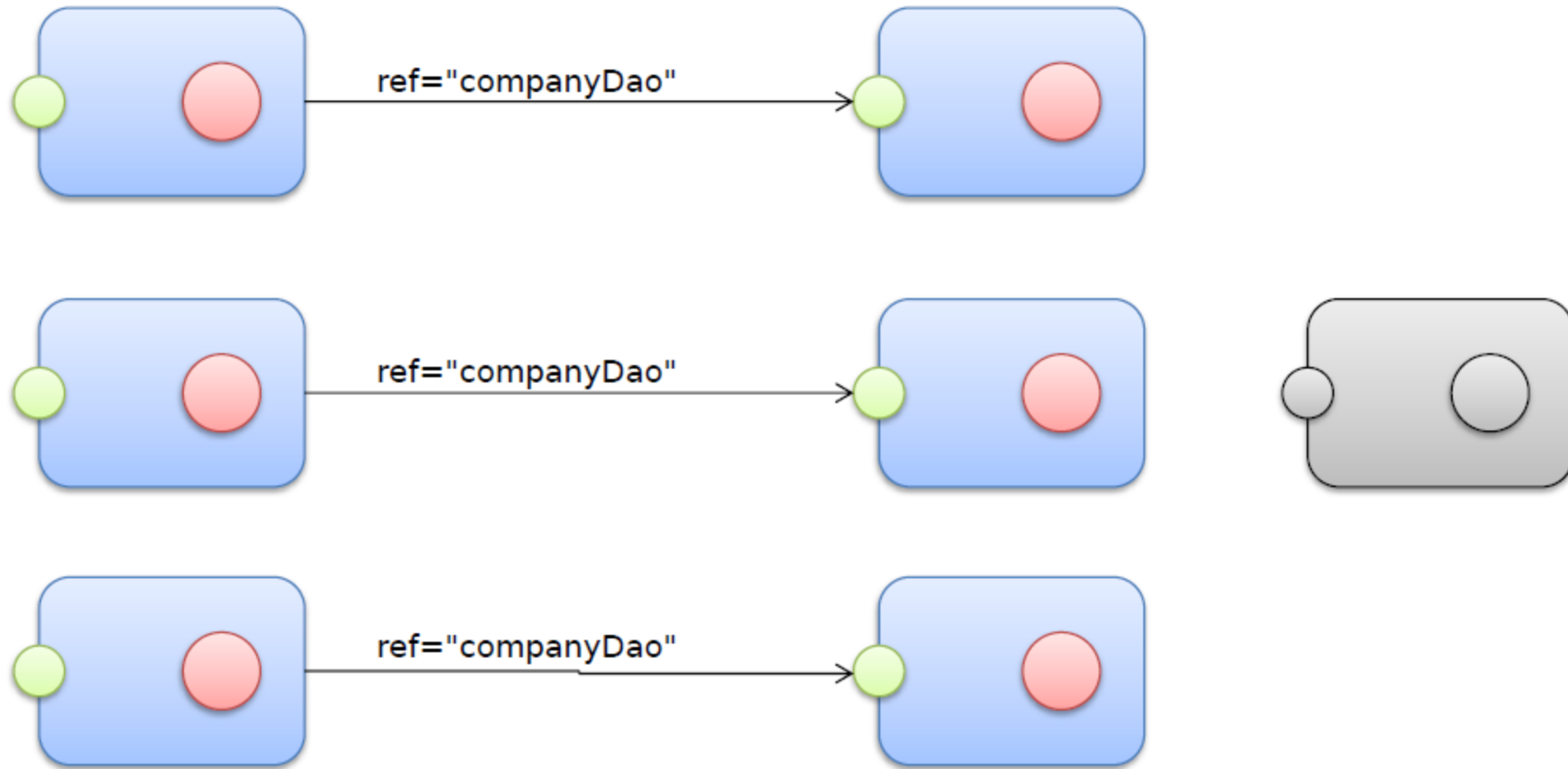
```
@Service("companyService")  
@Scope("prototype")  
public class CompanyServiceImpl implements CompanyService {  
...  
}
```



Singleton



Prototype



Custom 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();
}
```



Custom scope – rejestracja i użycie

```
<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>
```

```
<bean id="scopedCompanyService"
      class="lab.java.spring.beans.CompanyServiceImpl"
      scope="myScope">

  <aop:scoped-proxy proxy-target-class="false"/>

</bean>
```




```
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();  
}
```



Resource - implementacje

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



- *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");`



Resource 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"/>
```



Spring Container

Konfigurowanie kontenera i komponentów za pomocą kodu Java



- Adnotacją @Bean można oznaczyć metodę w klasie komponencie (tj. factory method)
- Zwracany obiekt zostaje umieszczony w kontenerze Spring
- W adnotacji można m.in. określić nazwę dla komponentu, scope itd.

```
@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() {
    // uwaga na kolejność !!
    return new CompanyServiceImpl(getCompanyDao());
}
```



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

```
@Configuration  
public ApplicationConfiguration{  
}
```



@Configuration – przykład użycia

@Configuration

```
@ComponentScan(  
    basePackages = {"com.trainings"},  
    excludeFilters = {  
        @ComponentScan.Filter(Controller.class)  
    })  
@ImportResource("classpath:spring.xml")  
public class AppConfig {
```

@Bean

```
    public JpaTransactionManager transactionManager() throws ClassNotFoundException {  
        return new JpaTransactionManager();  
    }  
}
```



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);
```



Inicjalizacja w kontekście webowym

```
<servlet>
  <servlet-name>webservice</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextClass</param-name>
    <param-value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>
  </init-param>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <!--<param-value>WebConfig, AppConfig</param-value> -->
    <param-value>com.trainings.web.springconfig</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

@Bean – więcej opcji

@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



Basepoint

@Bean – więcej opcji

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

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



- 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{
}
```



`@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>
```



Pobieranie właściwości

@Configuration

@PropertySource("/lab/spring/jdbc.properties")

```
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);  
    }  
}
```



Spring Container

Zaawansowany konfiguracja.



- 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`



- Własna fabryka, implementacja FactoryBean
- 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

- 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.

```
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {  
    @Override public void postProcessBeanFactory(  
        ConfigurableListableBeanFactory beanFactory) throws BeansException { }  
}
```



Wbudowane BeanFactoryPostProcessor

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



- 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



PropertyPlaceholderConfigurer

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations" value="classpath: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:hsqldb://production:9002

jdbc.username = sa

jdbc.password = root



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) {  
    }  
}
```



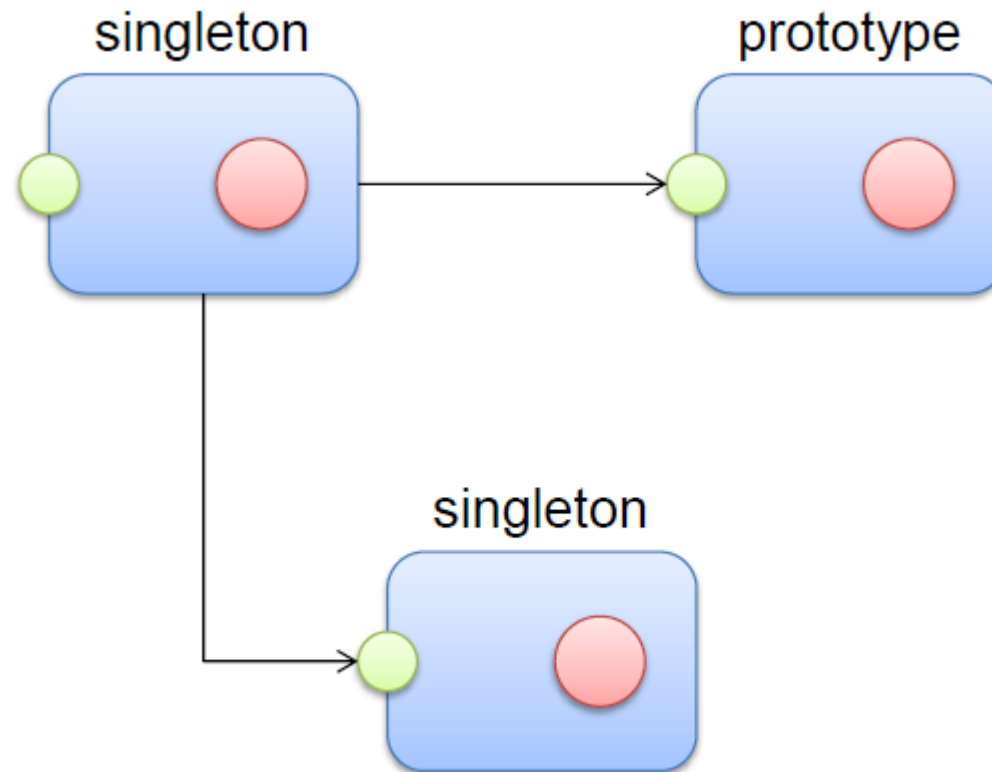
ApplicationContextAware

```
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..



Zależności – potencjalny problem

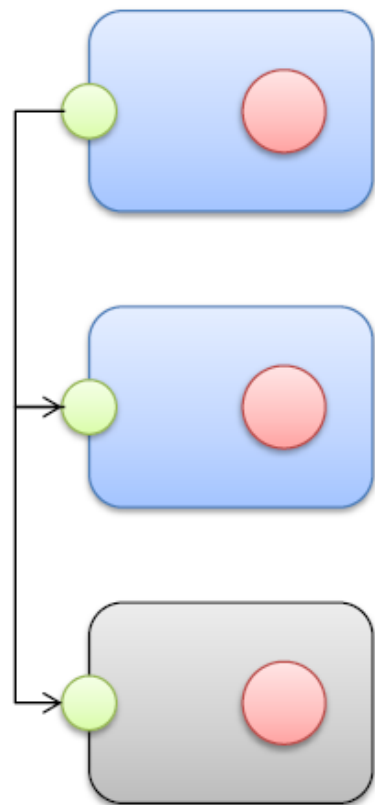
- Rozwiązanie pierwsze
 - Można skorzystać z ApplicationContextAware.
 - Wymaga to wprowadzenia zależności kodu od elementów Springa.
 - Wymaga to podania nazwy komponentu w kodzie i nie jest to już IoC
- Rozwiązanie drugie
 - Kontener może dostarczyć implementację zadanej metody w naszym kodzie.
 - Implementowana metoda będzie zwracała komponent tak jak skonfigurowane w pliku XML
 - Metoda musi mieć postać - <public|protected> [abstract] <zwracany-typ> nazwaMetody();

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

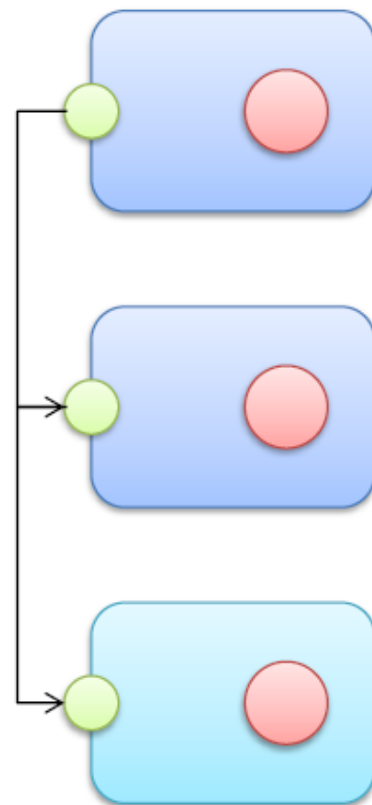

Konfiguracja XML

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





środowisko
"produkcyjne"



środowisko
programistyczne

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>
```

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

-Dspring.profiles.active=profile1, profile2



Testowanie

Spring testing, Junit.





Basepoint

Spring AOP

Programowanie aspektowe (Spring AOP, AspectJ)

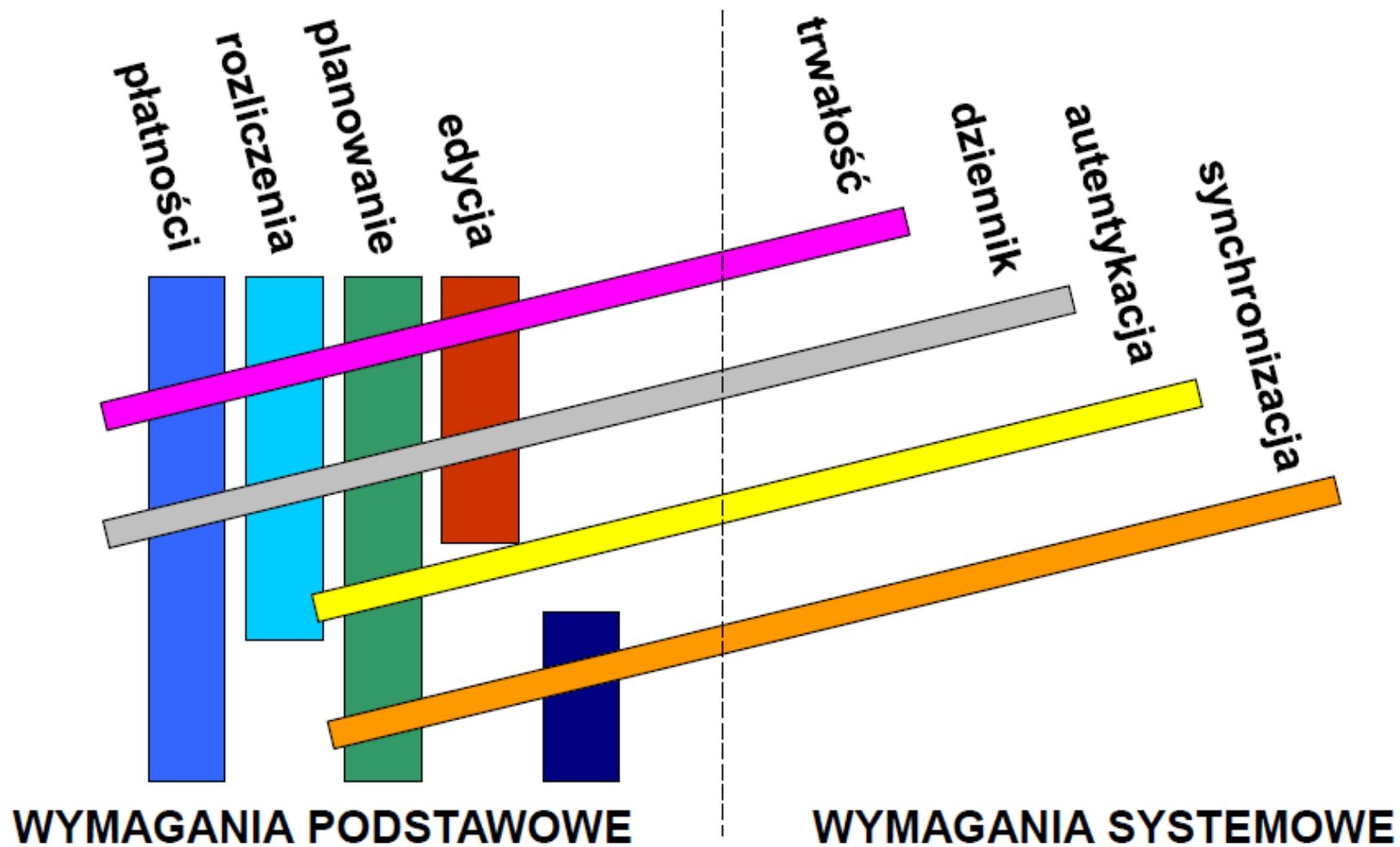


- Aspect Oriented Programming – programowanie aspektowe – próba rozwiązania ograniczeń pojawiających się przy stosowaniu analizy, projektowania i programowania zorientowanego obiektowo do złożonych problemów.
- Programowanie zorientowane aspektowo wraz z koncepcją komputerowej refleksji należy do metod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

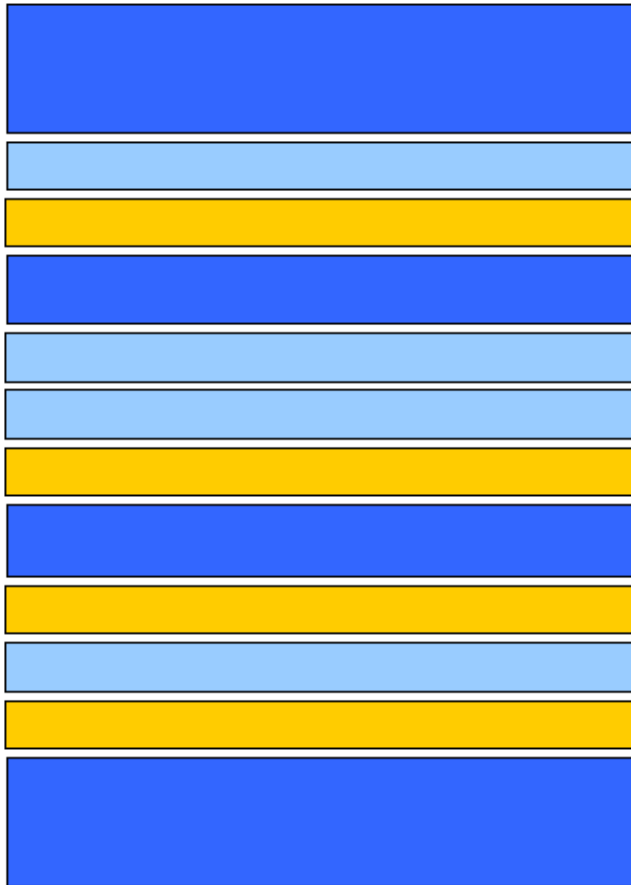
- Modularyzacja jest podstawą prawidłowej pielęgnacji kodu, pisali najwięksi badacze inżynierii oprogramowania. Postulowali oni dekompozycję programu na części, z których każda będzie dotyczyła jednego zagadnienia.
- Kolejne paradygmaty programowania, począwszy od programowania funkcyjnego, poprzez strukturalne,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



Podział wymagań w aplikacji



Modularyzacja kodu



PROGRAM OBIEKTOWY



PROGRAM ASPEKTOWY

- 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**

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



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

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



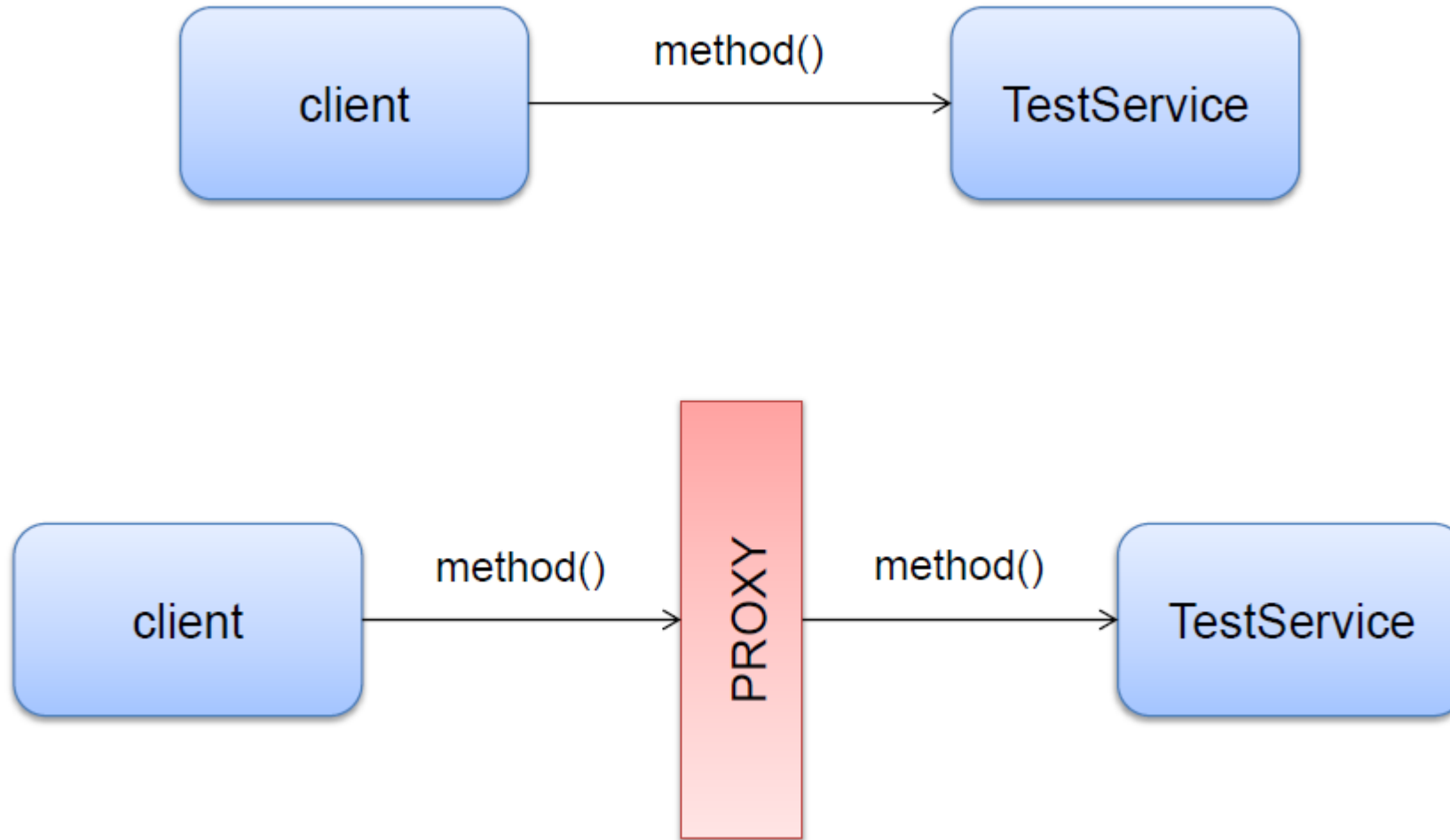
- **Wprowadzenie** (ang. introduction) to proces, który umożliwia modyfikację struktury obiektu przez wprowadzenie dodatkowych metod lub pól.

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



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

Proxy



Koncepcja proxy - przykład

```
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;  
    }  
}
```



Koncepcja proxy - przykład

```
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");  
}
```



```
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");  
    }  
}
```



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>
```



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



Porady, interceptor

```
public interface MethodBeforeAdvice extends BeforeAdvice {  
    void before(Method m, Object[] args, Object target) throws Throwable;  
}
```

```
public class RemoteThrowsAdvice implements ThrowsAdvice {  
    public void afterThrowing(RemoteException ex) throws Throwable {}  
}
```

```
public interface AfterReturningAdvice extends Advice {  
    void afterReturning(Object returnValue, Method m, Object[] args, Object target) throws Throwable;  
}
```

```
public class InvokeInterceptor implements MethodInterceptor {  
    public Object invoke(MethodInvocation invocation) throws Throwable { return null; }  
}
```



AOP w oparciu i schema

```
<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">
```

```
<aop:config>
```

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

```
</aop:config>
```

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

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

```
<!-- <aop:after/> <aop:after-returning/> <aop:after-throwing/> <aop:around/> -->
```

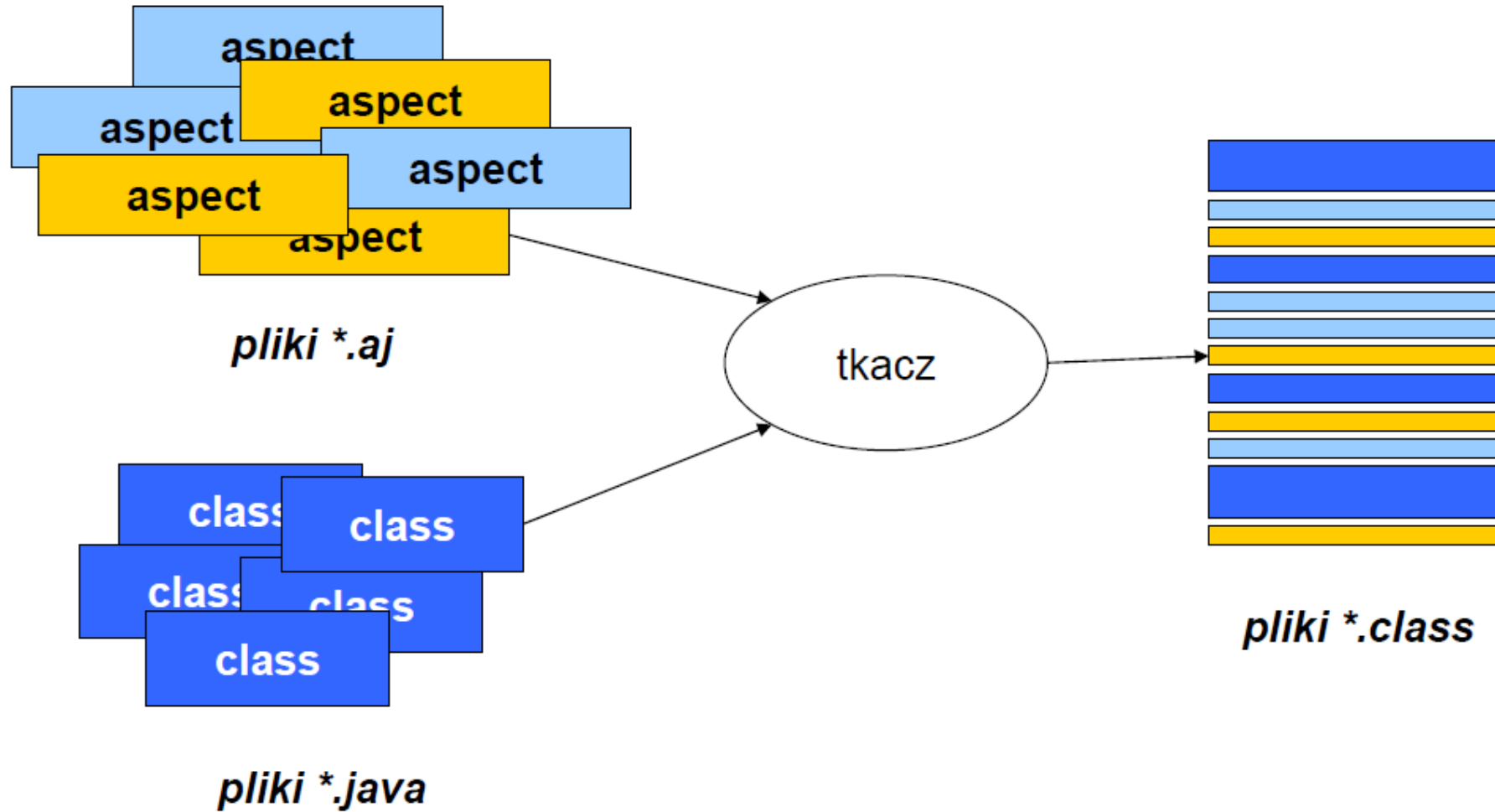
```
<bean id="aBean" class="..."> </bean>
```



- 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 (tkacz)



Adnotacja @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 {  
}
```



- 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

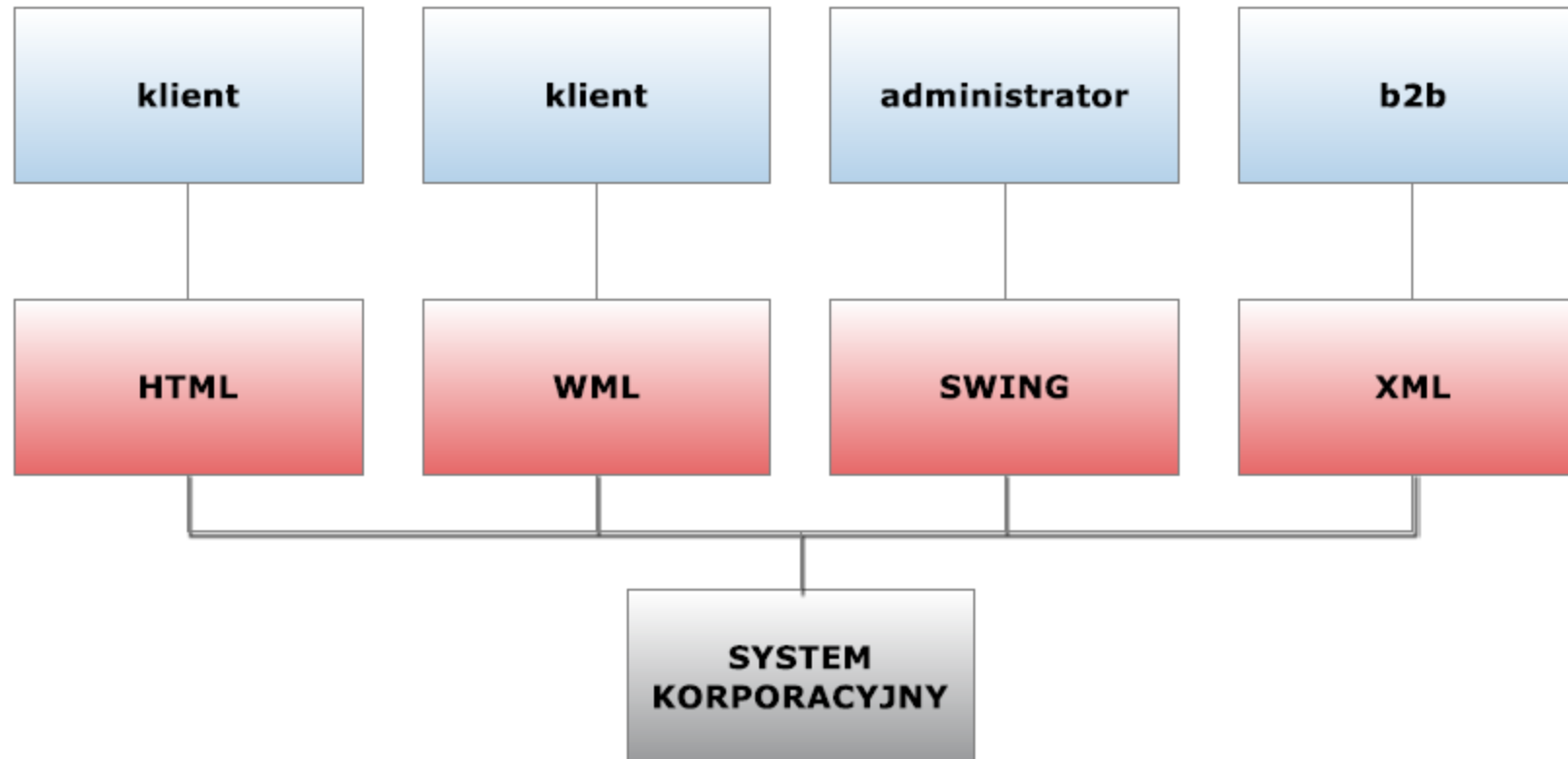


Aplikacje sieciowe

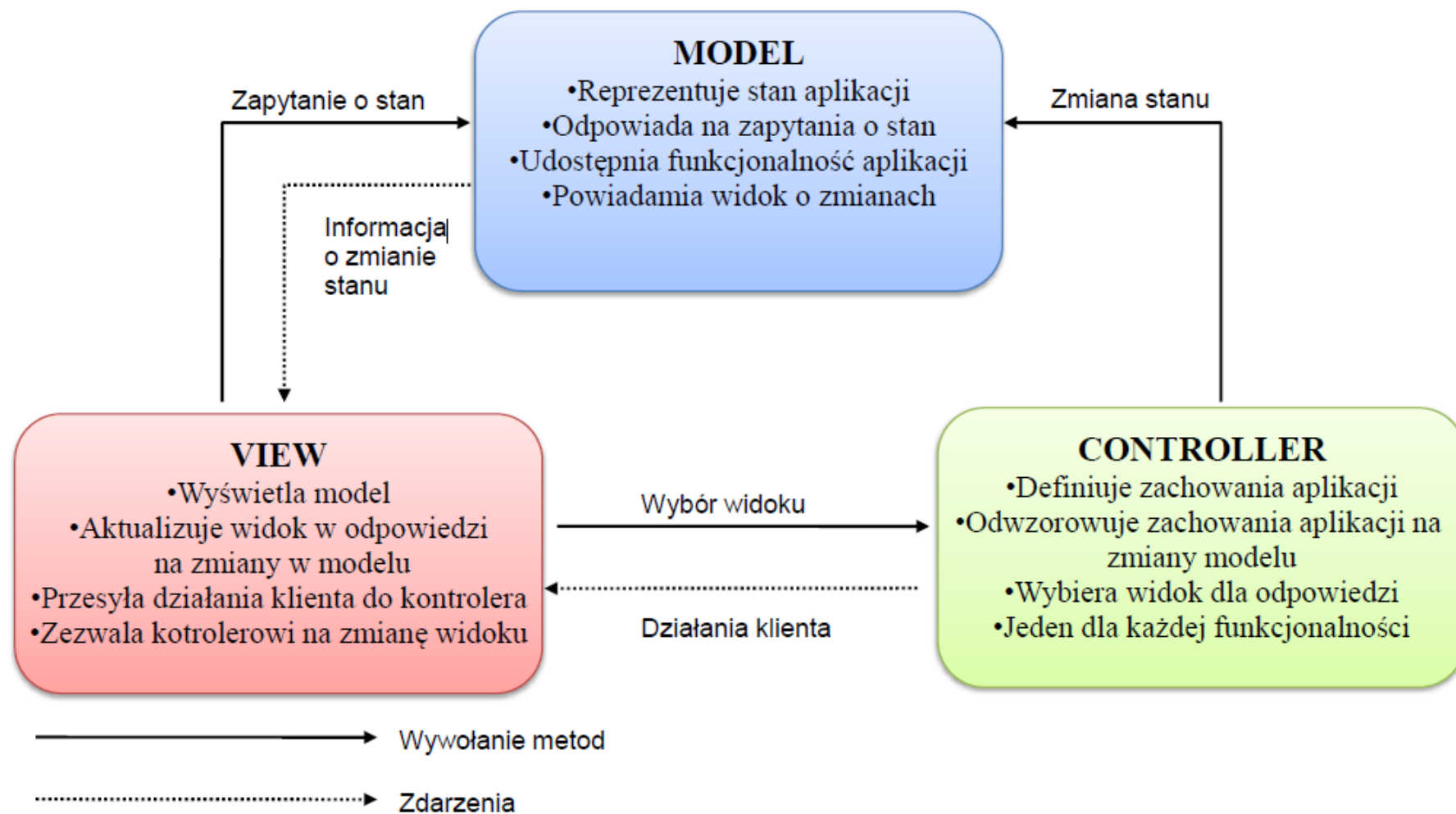
MVC. Spring w aplikacjach sieciowych. Serwlety. REST. Ajax.



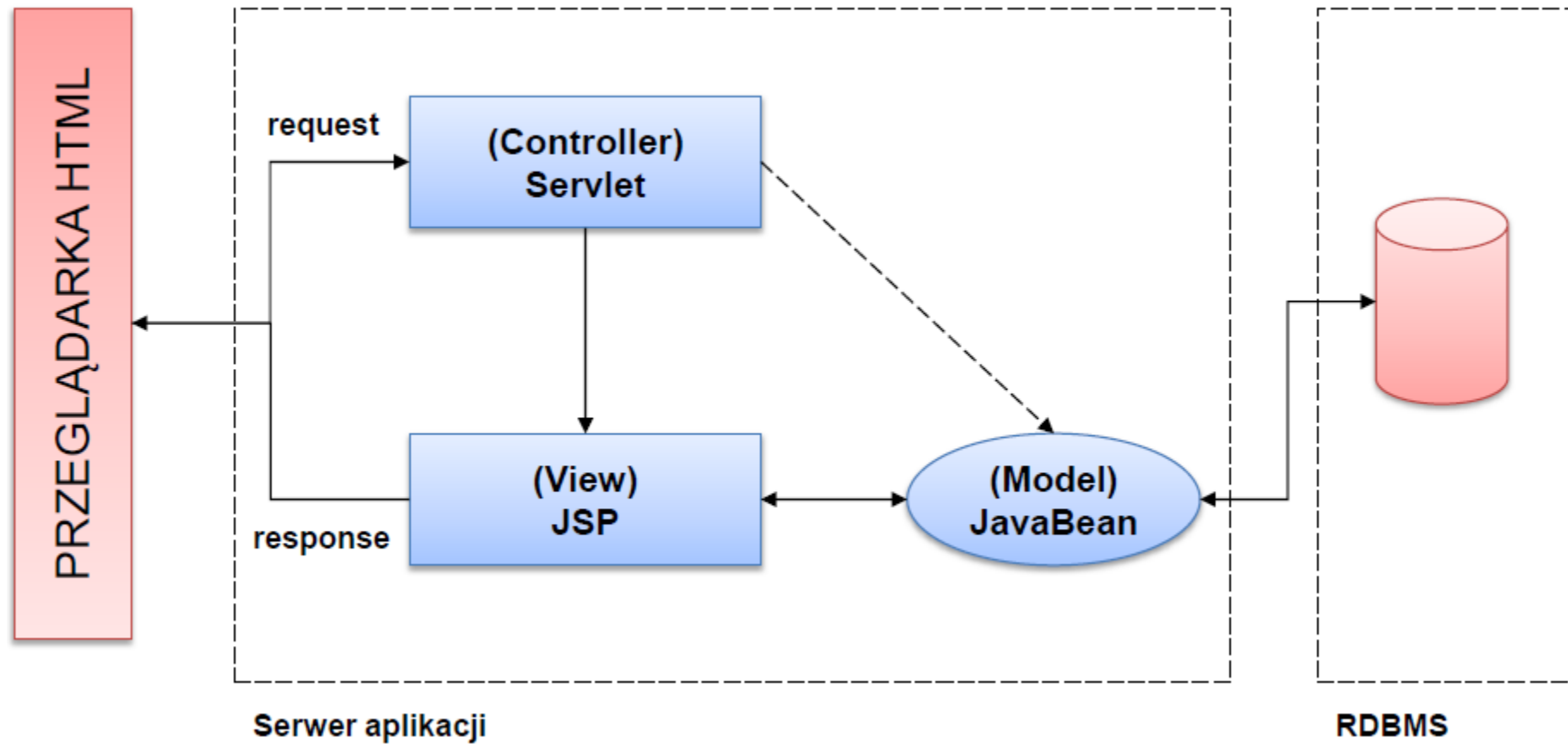
Aplikacja z różnymi interfejsami użytkownika



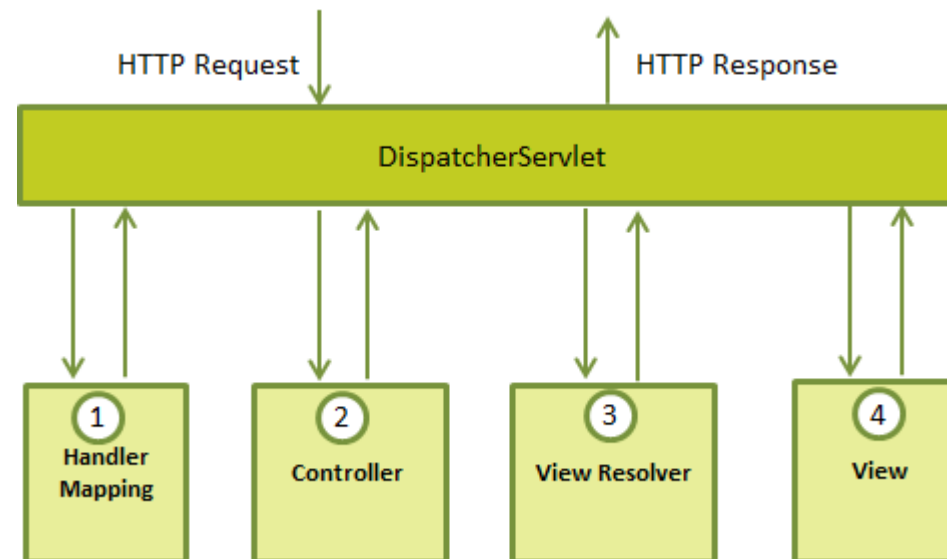
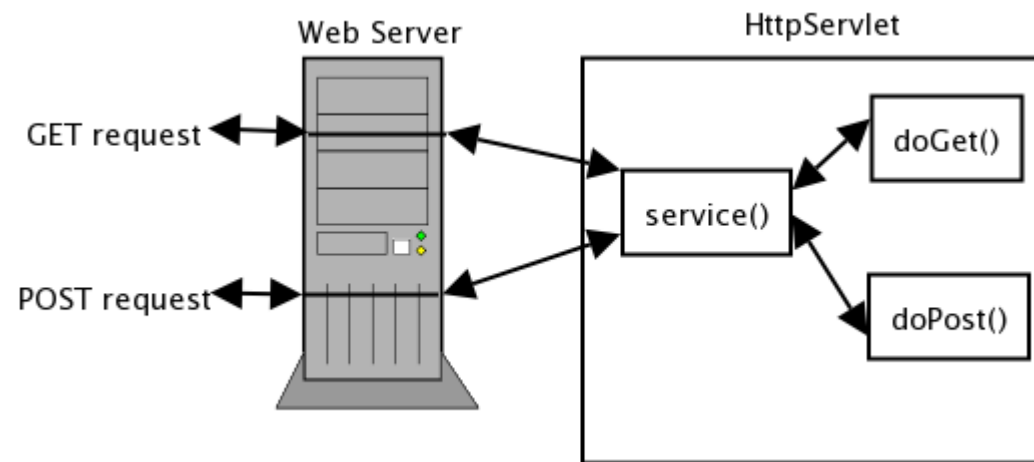
Wzorzec MVC



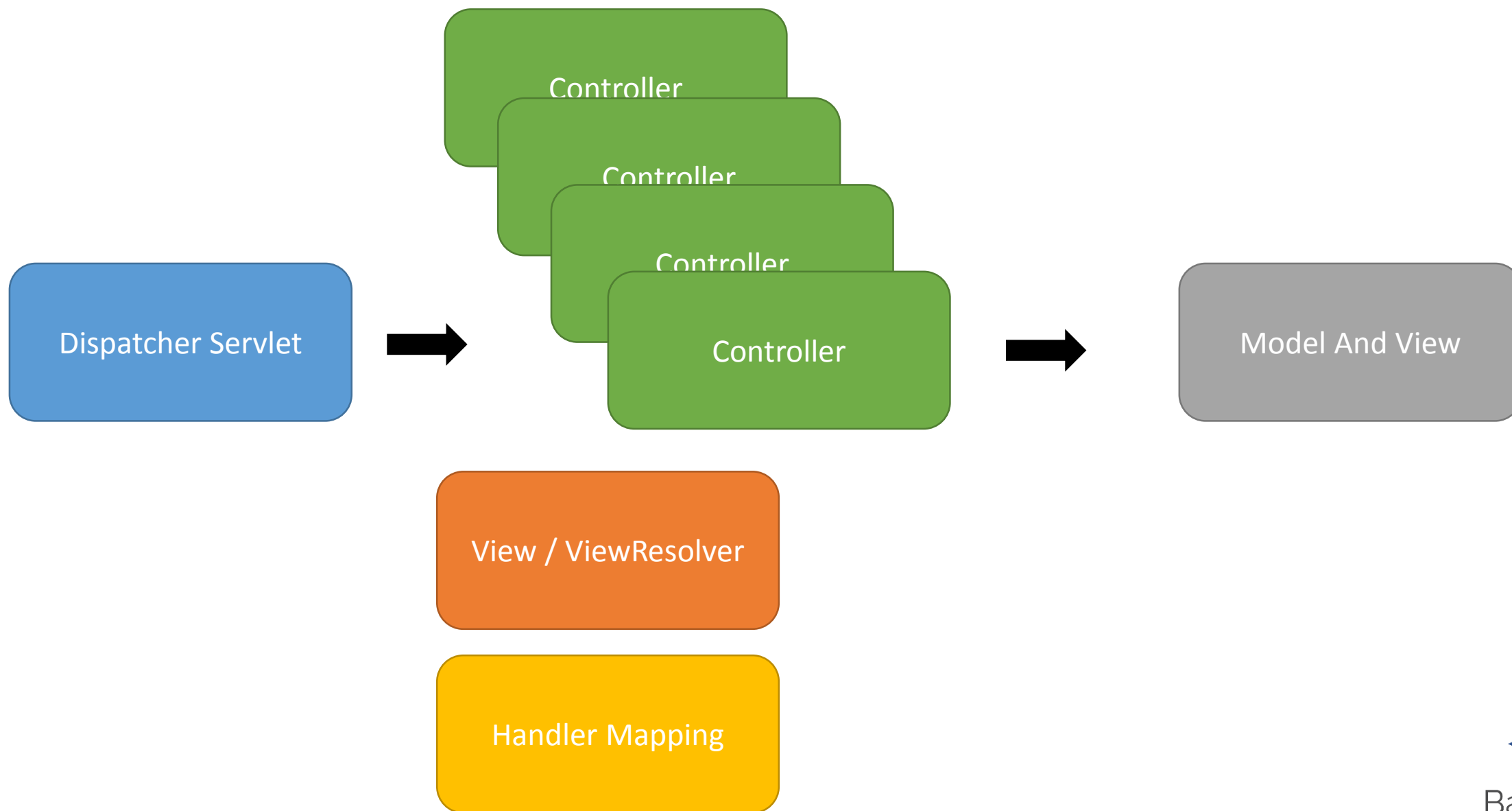
MVC w środowisku JEE



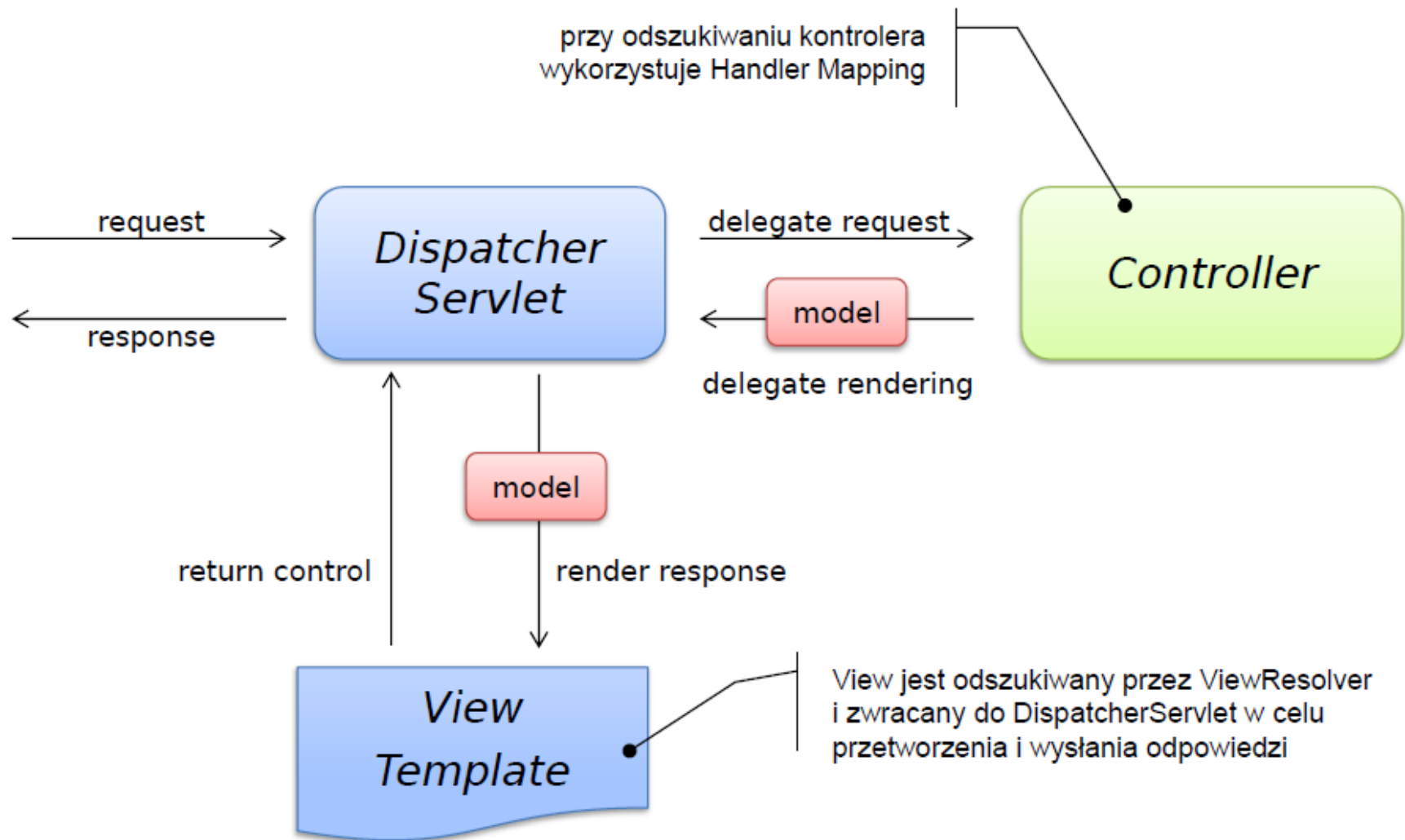
Server WEB



Przetwarzanie zapytań web w Spring MVC



Przebieg 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>
<init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>com.trainings.AppConfig</param-value>
</init-param>
</servlet>

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



Servlet API 3.0+

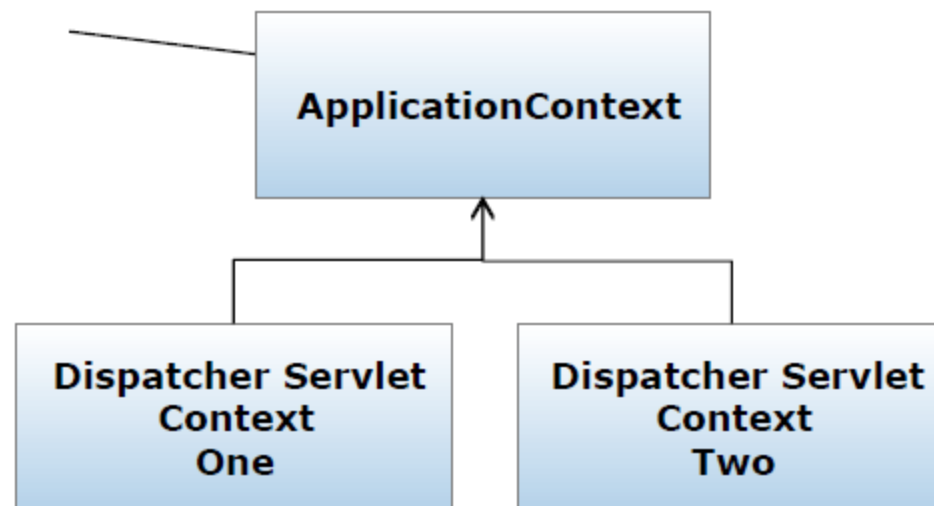
```
import org.springframework.web.WebApplicationInitializer;

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("*.html");
    }
}
```



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

```
<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 nazewnicznej /WEB-INF/**dispatcher**-servlet.xml



- W pierwszych wersjach Spring MVC istniała obiektowa hierarchia kontrolerów. Spring 3.0 uznaje ją za deprecated.
- W zamian wprowadza mechanizm definiowania właściwości kontrolera za pomocą adnotacji **@Controller**
- Od Spring 4.0 Spring wprowadza również adnotację **@RestController**



Adnotacja @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 kontrollerów

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

<mvc:annotation-driven />

<context:component-scan base-package="com.trainings.spring.web" />

@Configuration

@ComponentScan(basePackages = {"com.swapme.ws.controller"})

@EnableWebMvc

public AppConfig {}



- 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) {  
  
}
```



Konwertery i formater

```
<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 - 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>
```



- Interfejs HandlerInterceptor:
 - preHandle – zanim zostanie wywołana metoda kontrolera; jeśli zwróci false, żądanie nie jest dalej przetwarzane
 - postHandle – po wywołaniu metody kontrolera, ale zanim zostanie zrenderowany widok
 - afterCompletion – po zrenderowaniu 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; }  
  
    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; }  
    }  
}
```



Konfiguracja interceptora

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



Negocjacja zawartości (adnotacje)

@Configuration

@EnableWebMvc

```
public class WebConfig extends WebMvcConfigurerAdapter {
```

@Override

```
public void configureContentNegotiation( ContentNegotiationConfigurer configurer) {
```

```
    configurer.favorPathExtension(false).
```

```
    favorParameter(true);
```

```
} }
```



Negocjacja zawartości (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"/>



Basepoint

Resources

@Configuration

@EnableWebMvc

```
public class WebConfig extends WebMvcConfigurerAdapter {
```

@Override

```
public void addResourceHandlers( ResourceHandlerRegistry registry) {  
    registry.addResourceHandler("/resources/**").  
        addResourceLocations("/publicresources/").setCachePeriod(31556926);  
  
    registry.addResourceHandler("/resources/**")  
        .addResourceLocations( "/", "classpath:/META-INF/public-web-resources/")  
}  
}
```



Resources 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
```



Parametry @RequestMapping

- value – określa ścieżkę w URL
 - method – określa metodę GET/POST/...
 - params – parametry żądania HTTP (String[])
 - headers – nagłówki żądania HTTP (String[])
-
- Wszystkie elementy muszą 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.
 - `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`
 - obiekty adnotowane: `@PathVariable`, `@RequestParam`, `@RequestHeader`, `@RequestBody`, `@CookieValue`, `@MatrixVariable`
 - `java.util.Map`
 - `Command Object`




```
@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"; }
```

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



Ładne linki ("nice links") – 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 – zmienne tablicowe

- 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 zmiennych tablicowych (matrix variables)

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

```
@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET)
public void findPet(@PathVariable String 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
}
```



Obsługa matrix variables

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]
}
```



Metody kontrolera – zwracane wartości (1)

- ModelAndView – obiekt zawierający w sobie zarówno model, jak i nazwę widoku, który ma zostać wyświetlony
- Model – tylko model (zostanie użyty domyślny widok 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 – zwracane wartości (2)

- `@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

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

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:



Standardowe implementacje ViewResolver

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

Konfiguracja wielu resolverów widoków

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

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



Spring MVC

Techniki renderowania widoków, formularze.



Techniki renderowania widoków i tworzenia formularzy

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



- 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)



Biblioteka tagów do obsługi formularzy

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

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

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



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>
```

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

```
<form:form>
```

```
<form:errors path="*" cssClass="errorBox" />
```

```
<table> <tr>
```

```
    <td>First Name:</td>
```

```
    <td> <form:input path="firstName" /> </td>
```

```
    <td> <form:errors path="firstName" /> </td>
```

```
</tr> </table> </form:form>
```

```
public class Person {  
    @NotNull @Size(max=64) private String name;  
    @Min(0) @Max(110) private int age;  
}
```

- Validator:
 - 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 Validato



- 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



Spring MVC

Ajax. REST.



- Marshalling danych odbywa się na podstawie nagłówków
- W klasie adnotowanej jako `@Controller` definiujemy
 - metodę adnotowaną `@ResponseBody`
 - pola adnotowane `@RequestBody`
- Adnotacje te są zbędne gdy zastosujemy `@RestController` (Spring 4.0+)

Kontroller REST

```
@RequestMapping(value = "/contacts/{query}", produces = "application/json", method = RequestMethod.GET)
```

```
@ResponseBody public Contact queryContacts(  
    @PathVariable("query") String query) {  
    return contactService.listContactByName(query));  
}
```

```
@RequestMapping(value = "/contacts", produces = "application/json", method = RequestMethod.POST)
```

```
@ResponseBody public Contact addContact(@RequestBody Contact contact) {  
    return contactService.save(contact));  
}
```

```
@XmlRootElement("contacts")
```

```
static class Contact {}
```



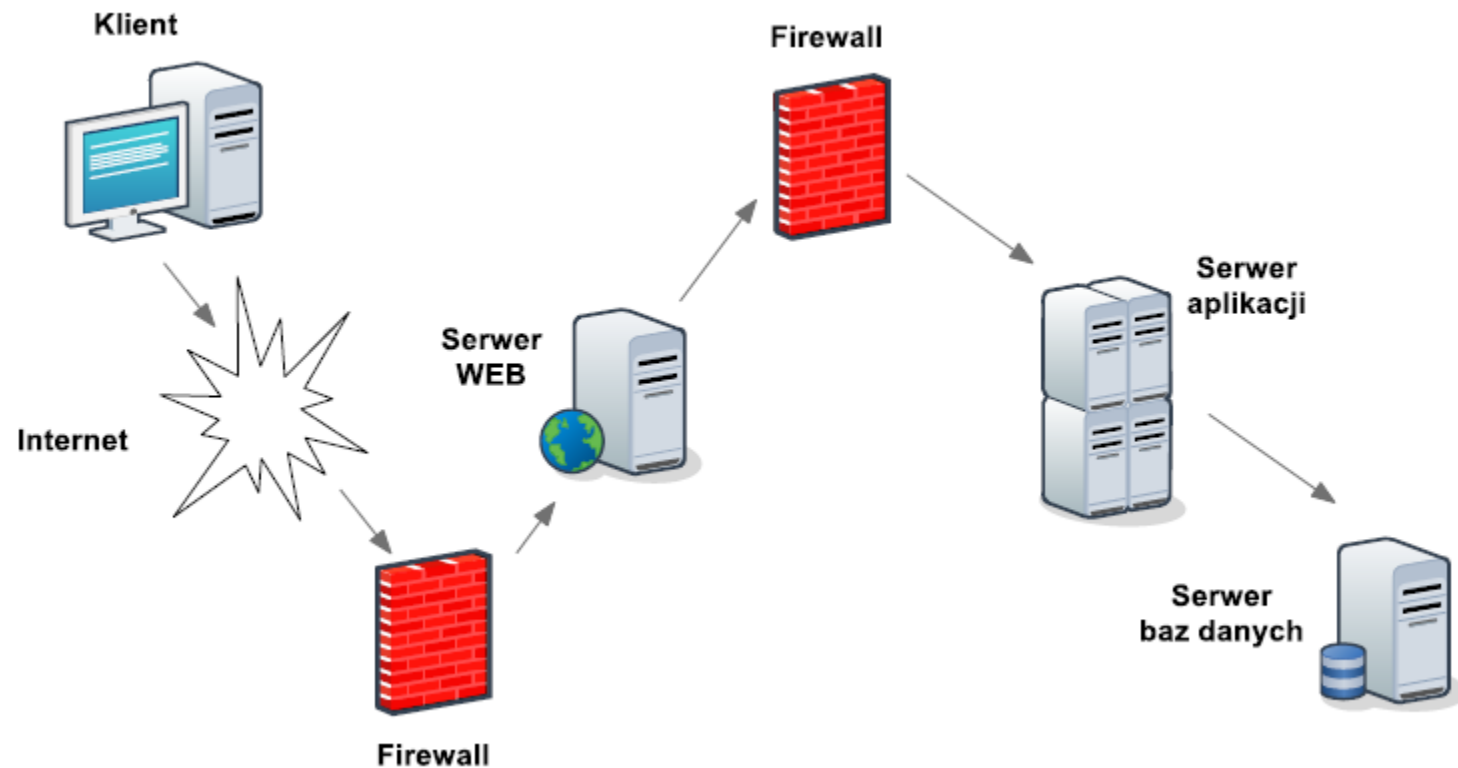
Spring security



Basepoint

Zabezpieczanie systemu komputerowego

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



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

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



Konfiguracija

```
<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" ....
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>
```



Konfiguracja spring-security.xml

```
<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="jimispasword" authorities="ROLE_USER, ROLE_ADMIN" />
    <security:user name="bob" password="bobspasword" 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>
```

Authentication manager

- Implementuje UserDetailsService

```
public interface UserDetailsService {  
    UserDetails loadUserByUsername(String var1) throws UsernameNotFoundException;  
}
```

```
<authentication-manager>  
<authentication-provider user-service-ref='myUserDetailsService'/>  
</authentication-manager>  
  
<bean id="myUserDetailsService" class="..."/>
```



Autentykacja w oparciu o bazę danych

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

```
<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>
```



Zabezpieczenia na poziomie bean'ów

```
<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>
```



Zabezpieczenie na poziomie bean'ów – adnotacja @Secured

```
@Service public class ContactServiceImpl implements ContactService {  
    @Autowired private ContactDAO contactDAO;  
  
    @Transactional @Secured(value = "ROLE_USER") public void addContact(Contact contact) {  
        contactDAO.addContact(contact);  
    }  
  
    @Transactional @Secured(value = "ROLE_USER") public List<Contact> listContact() {  
        return contactDAO.listContact();  
    }  
  
    @Transactional @Secured(value = "ROLE_ADMIN") public void removeContact(Integer id) {  
        contactDAO.removeContact(id);  
    }  
}
```

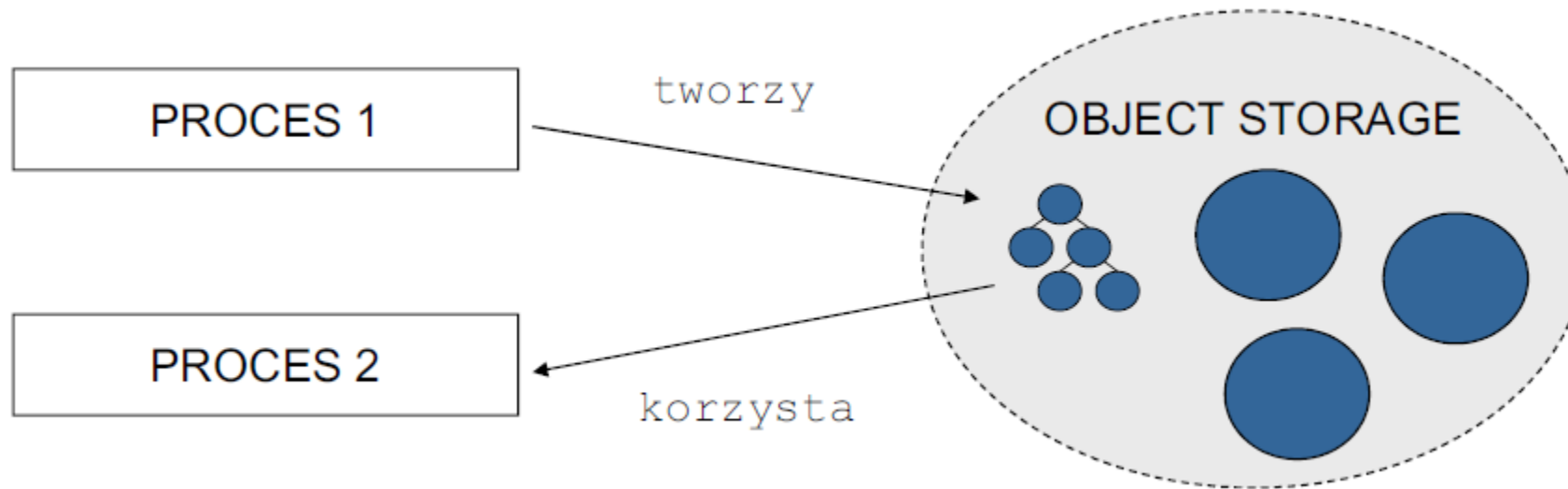
Integracja z JPA/Hibernate

Spring + wsparcie mapowania obiektowo-relacyjnego

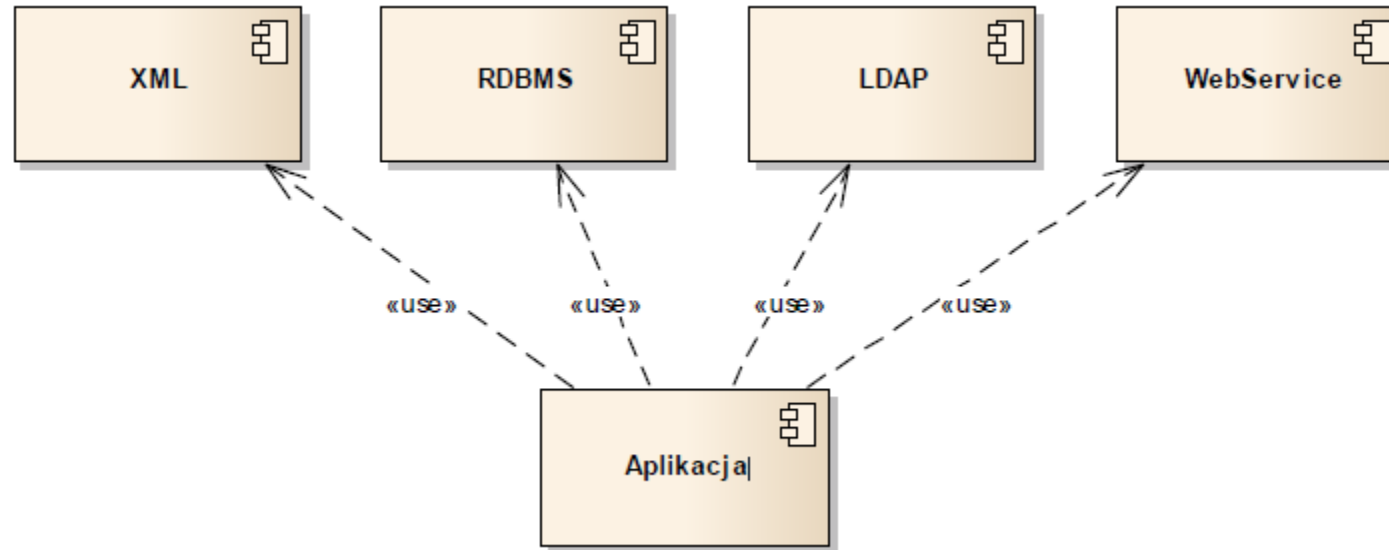


Trwałość obiektów

- W aplikacjach obiektowych mechanizmy trwałości pozwalają obiektom żyć dłużej niż proces, który je utworzył
- Stan obiektu zostaje zachowany np. na twardym dysku
- Zapis stanu dotyczy całego grafu powiązanych obiektów



- 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.

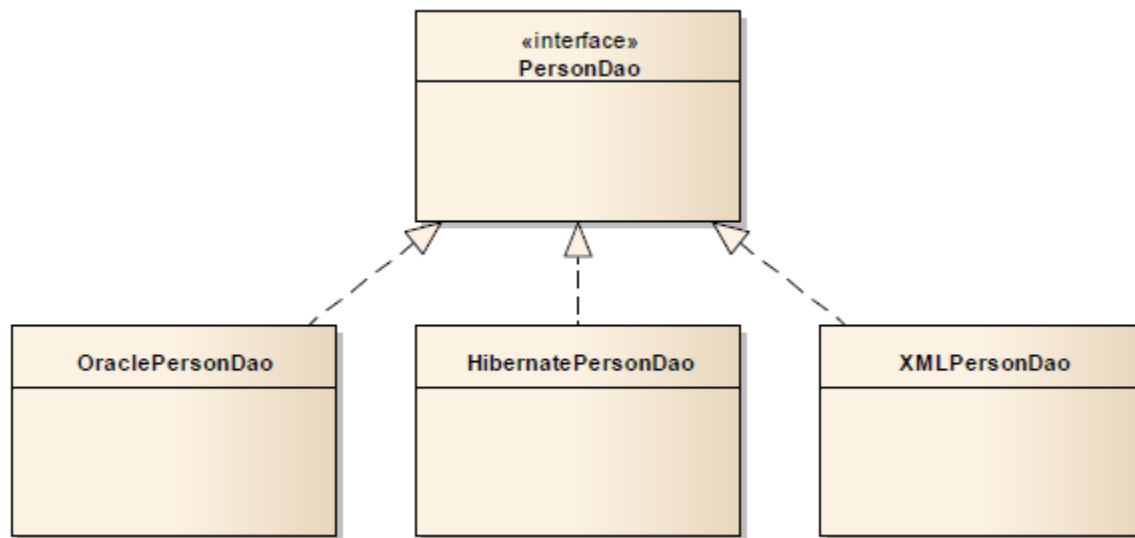


Mechanizmy trwałości

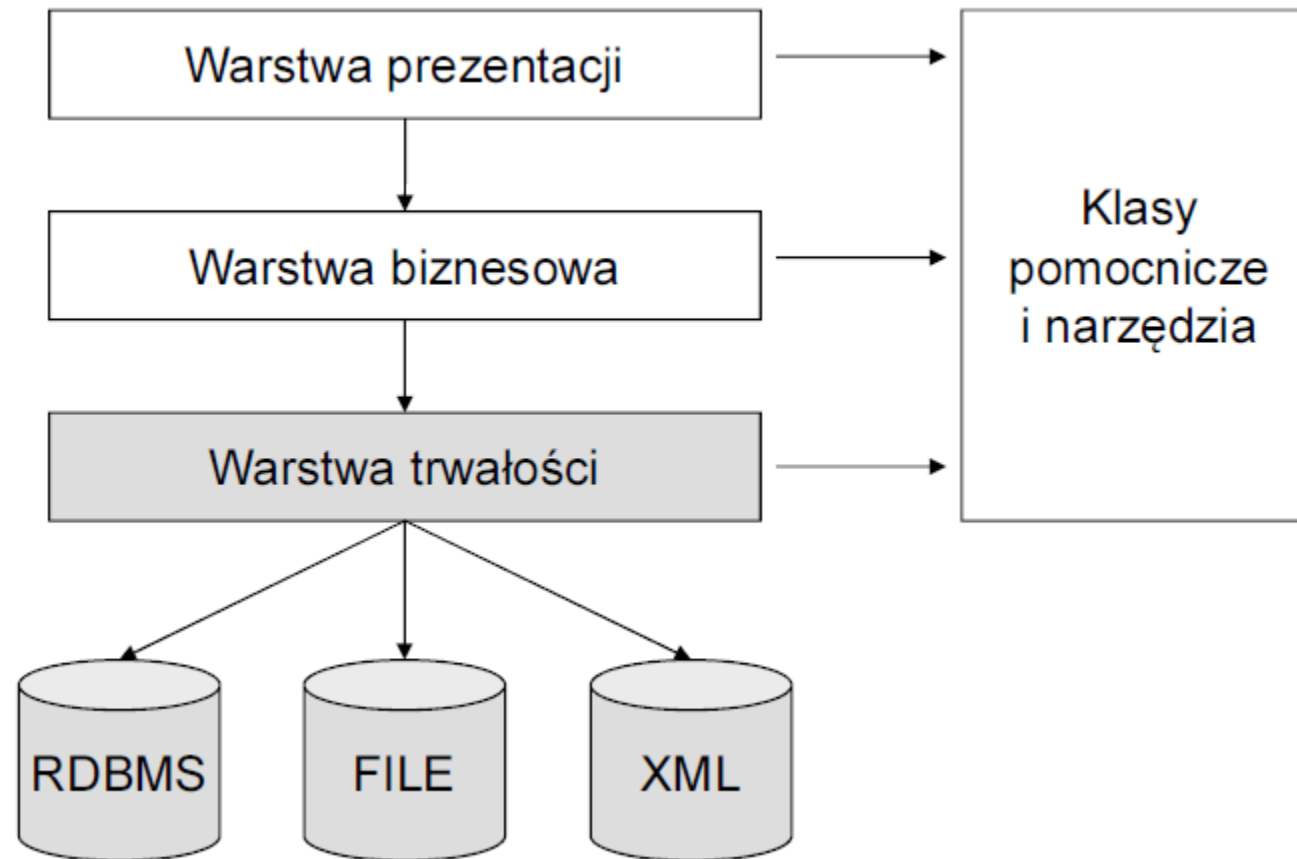
- Relacyjne bazy danych
- Obiektowe bazy danych
- Bazy NoSQL
- Bazy XML
- Serializacja



- 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



Architektura warstwowa



- 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
- HibernateTemplate
- JdoTemplate
- JpaTemplate
- SqlMapClientTemplate

JdbcDaoSupport

HibernateDaoSupport

JdoDaoSupport

JpaDaoSupport

SqlMapClientDaoSupport



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



Przykład DAO

```
public interface CompanyDao {  
    public Company get(Long id);  
    public List<Company> selectAll();  
    public void save(Company company);  
    public void delete(Company company);  
}
```

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

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



Konfiguracja DataSource w Springu

```
<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
 - **BoneCPDataSource**



Inicjalizacja EntityManagerFactory

- Utworzenie
 - pobranie z serwera aplikacyjnego
 - LocalEntityManagerFactoryBean
 - LocalContainerEntityManagerFactoryBean
- Pozyskanie 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

- 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

```
<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>
```



Pobranie 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/>
```



Spring TX

Transakcje w środowisku zarządzanym Spring.



- 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.



- Jednolite API
 - JTA
 - JDBC
 - Hibernate
 - JPA
 - JDO
- DTM (declarative TM)
- Integracja z warstwą modelu

Transakcje globalne

- Transakcje globalne - wiele źródeł (głównie relacyjne bazy danych, *message queues*) Używa JTA, w powiązaniu z serwerem aplikacyjnym
- Transakcje lokalne (np. na połączeniu JDBC)
- Wspólne API

- 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)

- 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 (transaction managers)

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



- Programatic
 - mała ilość tranzakcji
 - *TransactionTemplate*
- Deklaratywne (declarative)
 - zazwyczaj dobry wybór!
 - duża ilość tranzakcji
 - Transakcje określone 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 określone za pomocą adnotacji lub plików konfiguracyjnych.



Transakcje w Spring

- SPI (service provider)
- TransactionStatus jest powiązany z wątkiem

```
public interface PlatformTransactionManager {  
  
    TransactionStatus getTransaction(  
        TransactionDefinition definition) throws TransactionException;  
    void commit(TransactionStatus status) throws TransactionException;  
    void rollback(TransactionStatus status) throws TransactionException;  
}
```



Wsparcie transakcji w Spring

- Konfiguracja XML:
 <tx:annotation-driven/>

- Konfiguracja adnotacjami
 @EnableTransactionManagement

- Definicja managera transakcji
 @Bean

```
public JpaTransactionManager transactionManager()  
    throws ClassNotFoundException {  
    JpaTransactionManager txManager= new JpaTransactionManager();  
    txManager.setEntityManagerFactory(  
        entityManagerFactoryBean().getObject());  
    return transactionManager;  
}
```



Oznaczenie metody transakcyjnej

```
@Transactional public void callService() {  
    // implementation  
}
```

```
@Transactional(  
    propagation= Propagation.MANDATORY,  
    isolation= Isolation.READ_UNCOMMITTED,  
    timeout= 20000)  
public void callService() {  
    // implementation  
}
```



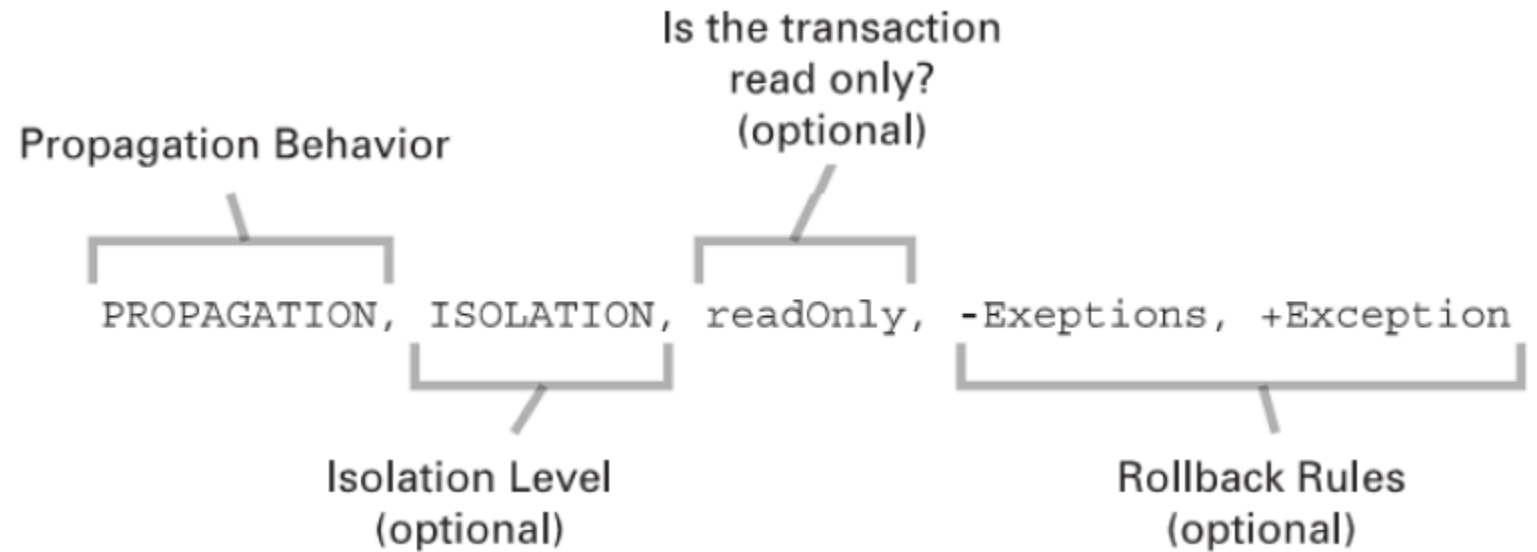
TransactionProxyFactoryBean

```
<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
- Warunki rollback



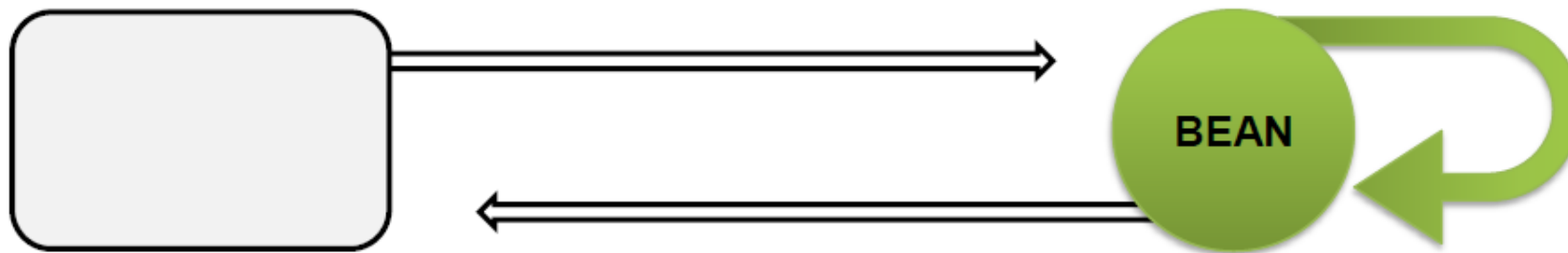
- **MANDATORY** – istniejąca, wyjątek jeśli nie istnieje
- **NESTED** – wykonaj w obrębie jeśli istnieje, w przeciwnym wypadku jak REQUIRED
- **NEVER** – bez transakcji, wyjątek gdy istnieje
- **NOT_SUPPORTED** – j.w., zawiesza transakcję
- **REQUIRED** – istniejąca albo nowa jeśli brak
- **REQUIRES_NEW** – nowa, zawieś jeśli istnieje
- **SUPPORTS** – w obrębie jeśli istnieje, nie transakcyjnie jeśli brak



PROPAGATION_REQUIRED

klient (komponent lub aplikacja)

Required



wątek wykonawczy



kontekst transakcyjny klienta



kontekst nietransakcyjny



kontekst transakcyjny komponentu

PROPAGATION_MANDATORY BEAN

klient (komponent lub aplikacja)

Mandatory



wątek wykonawczy



kontekst transakcyjny klienta



kontekst nietransakcyjny



kontekst transakcyjny komponentu

- **Read Uncommitted:** zezwala na *dirty reads*
- **Read Committed:** nie zezwala na *dirty read*
- **Repeatable Read:** wielokrotny odczyt zawsze daje ten sam wynik
- **Serializable:** wszystkie transakcje wykonywane sekwencyjnie

Isolation level / anomaly	Dirty reads	Non-repeatable reads	Phantoms Read
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Atrybuty adnotacji @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

