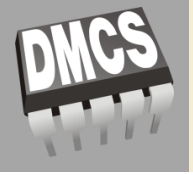


SPRING FRAMEWORK

Katedra Mikroelektroniki i Technik Informatycznych
Politechniki Łódzkiej
ul. Wólczanska 221/223 budynek B18,
90-924 Łódź

dr inż. Jakub Chłapiński

3. Spring Framework

Co to jest Spring Framework?

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Wielowarstwowy szkielet aplikacji Java/J2EE zawierający
  - Lekki kontener *IoC* umożliwiający scentralizowane zarządzanie i łączenie komponentów *JavaBean* i *POJO (Plain Old Java Object),*
  - Warstwę zarządzania transakcjami w oparciu o *AOP* (implementacja ograniczona do pojedynczego źródła danych),
  - Warstwę obsługi *JDBC*, oraz moduły umożliwiające integrację z bibliotekami *ORM* (Toplink, Hibernate, JDO, iBATIS) za pomocą ustandaryzowanych klas warstwy DAO,
  - Elastyczne środowisko do tworzenia aplikacji internetowych zgodnie z modelem MVC umożliwiające integrację ze Struts, WebWork, Tapestry i innymi
- Na całość szkieletu składa się około 20 modułów pogrupowanych w warstwy: Core Container, Data Access/Integration, Web, AOP, Instrumentation, Test

Warstwy i moduły w Spring

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Warstwa **Core Container** zawiera moduły *Core*, *Beans*, *Context* i *Expression Language*
  - *Core*, *Beans* – moduły implementujące lekki kontener obiektów z odwróconym sterowaniem oraz wstrzykiwaniem zależności. Moduły te zawierają implementację *BeanFactory*
  - *Context* – moduł umożliwiający pobranie obiektu z kontenera, z obługą internacjonalizacji, propagacją zdarzeń, ładowaniem zasobów, oraz dostęp do kontekstu aplikacji w zależności od typu aplikacji (np. poprzez *ServletContext*). Moduł ten zawiera implementację *ApplicationContext*
  - *Expression Language* – moduł implementujący język umożliwiający przeszukiwanie drzewa obiektów, pobieranie i ustawianie pól w obiektach, wywoływanie metod, operacje arytmetyczne oraz pobranie obiektów z kontenera IoC

Warstwy i moduły w Spring (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Warstwa **Data Access/Integration** zawiera moduły JDBC, ORM, OXM, JMS, Transaction
  - JDBC – moduł abstrakcji dla JDBC zawierający jednolite definicje błędów niezależnie od zastosowanego sterownika JDBC
  - ORM – moduł integracji z ORM API (Hibernate, JDO, iBatis, JPA)
  - OXM – moduł abstrakcji dla mapowania obiektów do XML przy użyciu JAXB, Castor, XMLBeans, JiBX, Xstream
  - JMS – implementacja Java Messaging Service API
  - Transaction – moduł do programowego oraz deklaratywnego zarządzania transakcjami

Warstwy i moduły w Spring (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Warstwa **Web** zawiera moduły Web, Web-Servlet, Web-Struts, Web-Portlet
  - Web – moduł umożliwiający integrację z podstawowymi funkcjonalnościami aplikacji internetowych takimi jak upload (wysyłanie do serwera) wieloczęściowego pliku oraz inicjalizację kontenera w środowisku Servlet API
  - Web-Servlet – moduł zawierający implementację abstrakcji wg wzorca MVC dla aplikacji opartej o Servlet API. Moduł ten umożliwia separację modelu domenowego od widoków oraz formularzy
  - Web-Struts – moduł integracji ze Struts 1.0. Obecnie moduł ten jest już przestarzały i nie należy go wykorzystywać w nowych aplikacjach
  - Web-Portlet – moduł dublujący funkcjonalność Web-Servlet do zastosowania w środowisku opartym o Portlet API

Warstwy i moduły w Spring (4)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Warstwa AOP zawiera implementację programowania aspektowego pozwalającą np. na definiowanie kodu przechwytującego wywołanie metod w klasach, zgodną ze standardem AOP-Alliance lub wykorzystującą AspectJ (moduł Aspects)
- Warstwa Instrumentation zawiera m.in. komponenty wspierające instrumentację aplikacji
- Warstwa Test zawiera wsparcie dla testowania aplikacji przy użyciu JUnit lub TestNG

Kontener IoC

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Obiekty którymi zarządza kontener IoC nazywamy bean-ami
- Aby aplikacja mogła działać, należy skonfigurować kontener IoC, m. in. wskazując które klasy zawierają definicje beanów z których składa się aplikacja; jakie występują pomiędzy nimi zależności oraz jak ma przebiegać zarządzanie ich czasem życia.
- Kontener IoC, reprezentowany w aplikacji przez interfejs org.springframework.context.ApplicationContext można skonfigurować na kilka sposobów:
  - Za pomocą pliku konfiguracyjnego w formacie XML
  - Za pomocą adnotacji w kodzie Java
  - Za pomocą kodu Java (konfiguracja programowa)
- Aby uruchomić kontener należy utworzyć przynajmniej jeden obiekt implementujący ApplicationContext

Konfiguracja

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Aby uruchomić kontener IoC należy utworzyć przynajmniej jeden obiekt implementujący ApplicationContext.
- W zależności od rodzaju aplikacji można wykorzystać różne implementacje ApplicationContext:
  - ClassPathXmlApplicationContext – implementacja poszukuje pliku konfiguracyjnego XML w ścieżce classpath

```
ApplicationContext context = new ClassPathXmlApplicationContext(new String[]
{"service.xml", "dao.xml", "web.xml"});
```

  - FileSystemXmlApplicationContext – implementacja poszukuje pliku konfiguracyjnego XML w dowolnym miejscu w systemie plików
  - WebXmlApplicationContext – implementacja poszukuje pliku konfiguracyjnego w „/WEB-INF/applicationContext.xml” oraz „/WEB-INF/???-servlet.xml”, gdzie ??? oznacza nazwę servletu

Konfiguracja (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Zastosowanie ContextLoaderListener w aplikacji internetowej

Plik /WEB-INF/web.xml

```
...

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/daoContext.xml
    /WEB-INF/applicationContext.xml
  </param-value>
</context-param>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

...
```

Konfiguracja (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykładowy plik konfiguracyjny XML

applicationContext.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-3.0.xsd">

<bean id="uzytkownikDao" class="my.application.dao.UzytkownikDao">
<!- konfiguracja oraz zaleznosci dla tego beanu -->
</bean>

...

<bean id=„uzytkownikService" class="my.application.service.UzytkownikService">
  <property name=„uzytkownikDao" ref=„uzytkownikDao"/>
  ...
</bean>

...

</beans>
```

Konfiguracja (4)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Konfiguracja z użyciem introspekcji

services.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"
  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:component-scan base-package="my.application.dao" />
  <context:component-scan base-package="my.application.service" />
</beans>
```

/my/application/service/UzytkownikService.java

```
package my.application.service;

@org.springframework.stereotype.Service(„uzytkownikService”)
public class UzytkownikService {

  @Autowired private UzytkownikDao uzytkownikDao;

  ...
}
```

Definicja beanu

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Najważniejsze atrybuty:
  - id – identyfikator beanu
  - class - klasa implementująca bean
  - scope – zasięg beanu
    - singleton (domyślnie) – oznacza iż kontener utworzy tylko jeden obiekt (instancję) danej klasy dla całej aplikacji
    - prototype – kontener przy każdym odwołaniu utworzy nową instancję
    - request (dla aplikacji www) – instancja będzie posiadała identyczny czas życia jak obiekt HttpServletRequest
    - session (dla aplikacji www) – instancja będzie posiadała czas życia jak obiekt HttpSession
    - global session (dla portletów) – czas życia tzw. sesji globalnej
  - init-method - metoda wywoływana przy inicjacji
  - destroy-method - metoda wywoływana tuż przed usunięciem instancji
  - lazy-init – instancja będzie tworzona dopiero przy pierwszym odwołaniu

Instancjacja beanu

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Instancjacja poprzez konstruktor (domyślnie) – wg specyfikacji JavaBean każda klasa definiująca bean powinna zawierać jeden bezparametrowy publiczy konstruktor. Domyślnie instancjacja beanu będzie przebiegać poprzez wywołanie bezparametrowego konstruktora.

Instancjacja beanu (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Instancjacja poprzez statyczna metodę

services.xml

```
...
<bean id="testService" class="my.application.service.TestService"
  factory-method="createInstance" />
...
```

/my/application/service/TestService.java

```
package my.application.service;

public class TestService {
  private static TestService INSTANCE = new TestService();
  private TestService() { }

  public static TestService createInstance() {
    return INSTANCE;
  }
  ...
}
```

Instancjacja beanu (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Instancjacja poprzez fabrykę instancji

services.xml

```
...
<bean id="testServiceFactory" class="my.application.service.TestServiceFactory" />

<!-- bez atrybutu class! -->
<bean id="testService"
  factory-bean="testServiceFactory"
  factory-method="createTestService" />
...
```

/my/application/service/TestServiceFactory.java

```
package my.application.service;

public class TestServiceFactory {
  public TestService createTestService() {
    return TestService.createInstance();
  }
  ...
}
```

Zależności

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Wstrzykiwanie zależności (*dependency injection*) do obiektu następuje w momencie jego instancjacji bądź zwrócenia przez statyczną metodę (*factory method*).
- Zależności mogą zostać wstrzyknięte do obiektu poprzez:
  - parametry konstruktora lub statycznej metody tworzącej
  - settery dla pól w klasie obiektu

Zależności (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład wstrzykiwania zależności przez parametry konstruktora

/my/application/service/UzytkownikService.java

```
package my.application.service;

public class UzytkownikService {
  private UzytkownikDao uzytkownikDao;

  public UzytkownikService(UzytkownikDao uzytkownikDao) {
    this.uzytkownikDao = uzytkownikDao;
  }

  ...
}
```

services.xml

```
...
<bean id="uzytkownikService" class="my.application.service.UzytkownikService">
  <constructor-arg ref="uzytkownikDao" />
</bean>
...
```

Zależności (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład wstrzykiwania zależności przez setter

/my/application/service/UzytkownikService.java

```
package my.application.service;

public class UzytkownikService {
  private UzytkownikDao uzytkownikDao;

  public void setUzytkownikDao(UzytkownikDao uzytkownikDao) {
    this.uzytkownikDao = uzytkownikDao;
  }

  ...
}
```

services.xml

```
...
<bean id="uzytkownikService" class="my.application.service.UzytkownikService">
  <property name="uzytkownikDao" ref="uzytkownikDao" />
</bean>
...
```

Konfiguracja poprzez adnotacje

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- W Spring począwszy od wersji 2.5 możliwe jest również skonfigurowanie zależności w klasach beanów poprzez zastosowanie adnotacji
- W tym celu należy jednak odpowiednio skonfigurować niezbędne mechanizmy postprocessingu w kontenerze np. poprzez plik XML
- Przykładowo poniższa konfiguracja spowoduje zarejestrowanie w kontenerze następujących postprocesorów:
  - AutowiredAnnotationBeanPostProcessor,
  - CommonAnnotationBeanPostProcessor,
  - PersistenceAnnotationBeanPostProcessor,
  - RequiredAnnotationBeanPostProcessor

```
<?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-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

@Required

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Adnotacja @Required stosowana przy setterach określa iż dana zależność musi zostać wstrzyknięta.

```
public class UzytkownikService {
  private UzytkownikDao uzytkownikDao;

  @Required //blad jezeli nie ustawimy tej zaleznosci w konfiguracji
  public void setUzytkownikDao(UzytkownikDao uzytkownikDao) {
    this.uzytkownikDao = uzytkownikDao;
  }
}
```

@Autowired i @Inject

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Adnotacja @Autowired lub @Inject określa iż dana zależność ma być wstrzyknięta automatycznie.

```
public class UzytkownikService {
  private UzytkownikDao uzytkownikDao;

  @Autowired(required=true)
  public void setUzytkownikDao(UzytkownikDao uzytkownikDao) {
    this.uzytkownikDao = uzytkownikDao;
  }
}
```

```
public class UzytkownikService {
  private UzytkownikDao uzytkownikDao;

  @Inject
  public void setUzytkownikDao(UzytkownikDao uzytkownikDao) {
    this.uzytkownikDao = uzytkownikDao;
  }
}
```

@Autowired (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Adnotację @Autowired można również stosować do innych metod, również z wieloma argumentami

```
public class UzytkownikService {
  private UzytkownikDao uzytkownikDao;

  @Autowired
  public void wyszukaj(UzytkownikDao uzytkownikDao, AdminDao adminDao) {
    ...
  }
}
```

@Autowired (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Adnotację @Autowired można również umieszczać przy polach klasy

```
public class UzytkownikService {

  @Autowired
  private UzytkownikDao uzytkownikDao;

  ...
}
```

@Autowired (4)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Adnotację @Autowired można również używać dla tablic i kolekcji w celu wyszukania wszystkich beanów danego typu

```
public class UzytkownikService {

  @Autowired
  private GeneryczneDao[] generyczneDao;

  @Autowired
  private Set<GeneryczneDao> generyczneDaoSet;

  @Autowired
  //jako klucz uzyte zostana nazwy beanow
  private Map<String, GeneryczneDao> generyczneDaoMap;

  ...
}
```

@Autowired i @Qualifier

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- W przypadku istnienia twóch beanów o jednakowym typie można doprecyzować o którą zależność chodzi przez adnotację @Qualifier

```
public class UzytkownikService {

  @Autowired
  @Qualifier("adminDao")
  private UniwersalneDao dao;

  ...
}
```

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

  <bean class="dao.UniwersalneDao">
    <qualifier value=„adminDao"/>
  </bean>

  <bean class="dao.UniwersalneDao">
    <qualifier value=„userDao"/>
  </bean>

  ...

</beans>
```

@Resource

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Adnotacja @Resource działa w sposób zbliżony do @Autowire, przy czym identyfikuje ona zależność po nazwie, a nie po typie
- W przypadku gdy nie została określona nazwa poprzez atrybut „name” adnotacja @Resource działa podobnie do @Autowired

```
public class UzytkownikService {

  @Resource(name="uzytkownikDao")
  private UzytkownikDao uzytkownikDao;

  ...
}
```

@PostContruct i @PreDestroy

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Adnotacje @PostConstruct i @PreDestroy umożliwiają wskazanie metod które zostaną wywołane bezpośrednio po instacjacji (@PostConstruct) i przed usunięciem beanu (@PreDestroy)

```
public class LicznikZyciaService {

  @PostConstruct
  public void start() {
    ...
  }

  @PreDestroy
  public void koniec() {
    ...
  }

  ...
}
```

Skanowanie klas beanów

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

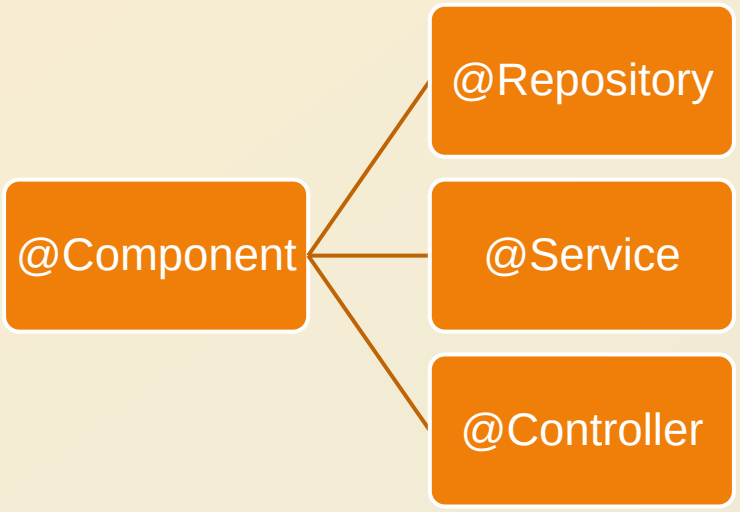
- Począwszy od wersji 2.5 Spring Framework posiada mechanizmy skanowania klas aplikacji (tzw. introspekcji) w poszukiwaniu definicji beanów
- Aby uruchomic skanowanie klas w aplikacji należy odpowiednio skonfigurować kontener

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

  <context:component-scan base-package="org.example.dao">
    <context:include-filter type="regex" expression=".*Dao"/>
    <context:exclude-filter type="annotation"
      expression="org.springframework.stereotype.Component"/>
  </context:component-scan>
  <context:component-scan base-package="org.example.service"/>

</beans>
```

@Component

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Aby dana klasa została dodana do kontenera musi być opatrzona adnotacją @Component lub pochodną




```
@Component("testowyBean")
public class TestowyBean {
  ...
}
```

@Repository

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Adnotacja @Repository rozszerza @Component określając tzw. stereotyp beanu jako obiekt warstwy dostępu do danych (DAO)
- Obecnie Spring wykorzystuje tą adnotację do automatycznej translacji wyjątków

```
@Repository(„uzytkownikDao")
public class UzytkownikDao {
  ...
}
```

@Service

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Adnotacja @Service rozszerza @Component określając tzw. stereotyp beanu jako obiekt warstwy logiki biznesowej (klasy serwisowej)
- Obecnie Spring nie wykorzystuje tej adnotacji w żaden szczególny sposób, ale zaleca się wyróżnianie klas serwisowych przy pomocy tej adnotacji.

```
@Service(„uzytkownikService")
public class UzytkownikService {
  ...
}
```

@Controller

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Adnotacja @Controller rozszerza @Component określając tzw. stereotyp beanu jako obiekt kontolera dla Spring MVC
- Obecnie Spring nie wykorzystuje tej adnotacji w żaden szczególny sposób, ale zaleca się jej stosowanie dla definicji kontrolerów

```
@Controller
@RequestMapping("/uzytkownik.htm")
@SessionAttributes("uzytkownik")
public class UzytkownikController {

  @RequestMapping(method = RequestMethod.GET)
  public String pokazFormularz(ModelMap model) {
    User user = new User();
    model.addAttribute(user);
    return "userForm";
  }

  @RequestMapping(method = RequestMethod.POST)
  public String zapiszUzytkownika(@ModelAttribute("uzytkownik") Uzytkownik uzytkownik) {
    ...
    return "redirect:uzytkownikZapisany.htm";
  }
}
```

@Scope

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Aby zdefiniować zasięg beanu należy zastosować adnotację @Scope

```
@Service
@Scope("prototype")
public class UzytkownikService {
  ...
}
```

Konfiguracja przy pomocy kodu Java

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Aby dodać do kontenera nowe definicje beanów można zastosować klasę oznaczonę adnotacją @Configuration

```
@Configuration
public class Konfiguracja {

  @Bean public UzytkownikService uzytkownikService() {
    return new UzytkownikService();
  }

  ...
}
```

```
<beans>
  <bean id=„uzytkownikService" class=„service.UzytkownikService"/>
</beans>
```

Konfiguracja przy pomocy kodu Java (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Do uruchomienia kontenera w oparciu o konfiguracje Java należy posłużyć się implementacją AnnotationConfigApplicationContext

```
public class Test {

  public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(Konfiguracja.class);
    UzytkownikService us = ctx.getBean(UzytkownikService.class);
    ...
  }
}
```

```
ApplicationContext ctx = new AnnotationConfigApplicationContext(UzytkownikService.class,
    AdminService.class, UzytkownikDao.class);
UzytkownikService us = ctx.getBean(UzytkownikService.class);
```

```
ApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.register(UzytkownikService.class, AdminService.class, UzytkownikDao.class)
ctx.refresh();
UzytkownikService us = ctx.getBean(UzytkownikService.class);
```

Konfiguracja przy pomocy kodu Java (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykładowa konfiguracja dla aplikacji internetowej

/WEB-INF/web.xml

```
<web-app>
  <context-param>
    <param-name>contextClass</param-name>
    <param-value>
      org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
  </context-param>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>my.application.Konfiguracja</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
```

Konfiguracja przy pomocy kodu Java (4)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykładowa konfiguracja dla aplikacji internetowej (c.d)

```
  <servlet>
    <servlet-name>dispatcher</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>my.application.KonfiguracjaWeb</param-value>
    </init-param>
  </servlet>

  ...
</web-app>
```

Konfiguracja przy pomocy kodu Java (5)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Klasa konfigurująca może dołączać konfiguracje zdefiniowane przez inne klasy przy pomocy adnotacji @Import

KonfiguracjaDao.java

```
@Configuration
public class KonfiguracjaDao {
  public @Bean TestDao testDao() {
    return new TestDao();
  }
  ...
}
```

KonfiguracjaService.java

```
@Configuration
public class KonfiguracjaService {
  public @Bean TestService testService() {
    return new TestService();
  }
  ...
}
```

Konfiguracja.java

```
@Configuration
@Import(KonfiguracjaDao.class)
@Import(KonfiguracjaService.class)
public class Konfiguracja {
...
}
```

Łączenie metod konfiguracji (1)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Dołączanie konfiguracji w Java do konfiguracji w XML

beans.xml

```
<beans>
  <!-- introspekcja klas z adnotacjami np. @Autowired, @Configuration -->
  <context:annotation-config/>

  <!- klasa konfigurująca jest zwyklym beanem -->
  <bean class="app.Konfiguracja"/>

  <!- klasa konfigurująca moze tez zostac dolaczona przez introspekcje -->
  <context:component-scan base-package="app"/>

  ...
</beans>
```

app/Konfiguracja.java

```
@Configuration
public class Konfiguracja {
...
}
```

Uruchomienie kontenera w kodzie

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
```

Łączenie metod konfiguracji (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Dołączanie konfiguracji w XML do konfiguracji w Java

app/Konfiguracja.java

```
@Configuration
@ImportResource("classpath:/beans.xml")
public class Konfiguracja {
...
}
```

beans.xml

```
<beans>
  ...
</beans>
```

Uruchomienie kontenera w kodzie

```
ApplicationContext context =
                    new AnnotationConfigApplicationContext(Konfiguracja.class);
```

Internacjonalizacja

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Interfejs ApplicationContext rozszerza interfejs MessageSource
- MessageSource zawiera metody:
  - `String getMessage(String code, Object[] args, String default, Locale locale)`
  - `String getMessage(String code, Object[] args, Locale locale) throws NoSuchMessageException`
  - `String getMessage(MessageSourceResolvable resolvable, Locale locale)`
- Po załadowaniu ApplicationContext w konfiguracji wyszukiwany jest automatycznie bean o nazwie "messageSource"
- W przypadku znalezienia odpowiedniego beanu wywolania w/w metod w ApplicationContext przekierowane zostaną do metod w beanie
- W przypadku gdy nie zostanie znaleziony odpowiedni bean metody przekierowywane są do pustego DelegatingMessageSource

Internacjonalizacja (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Spring dostarcza zasadniczo dwu różnych implementacji interfejsu MessageSource: ResourceBundleMessageSource i StaticMessageSource
- Obydwie implementacje oparte są o interfejs HierarchicalMessageSource, co pozwala na stosowanie hierarchicznej struktury obiektów MessageSource
- StaticMessageSource - rzadko używana implementacja pozwalająca na modyfikację wielojęzycznych komunikatów z poziomu programu
- ResourceBundleMessageSource - najczęściej stosowana implementacja wykorzystująca komunikaty wielojęzyczne zapisane w plikach .properties

Internacjonalizacja (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład konfiguracji ResourceBundleMessageSource

beans.xml

```
<beans>

<!-- w ponizszym przykladzie komunikaty beda wyszukiwane w plikach errors.properties,
warnings.properties i messages.properties (lub ich odpowiednich wersjach dla
poszczegolnych lokalizacji) lezacych w classpath -->

  <bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
      <list>
        <value>errors</value>
        <value>warnings</value>
        <value>messages</value>
      </list>
    </property>
  </bean>

</beans>
```

Internacjonalizacja (4)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykładowe pliki .properties z komunikatami

errors.properties

```
# bledy krytyczne
error.critical.required=The '{0}' property is required.

# inne bledy
typeMismatch.java.util.Date=Please enter a date in the format YYYY-MM-DD.
typeMismatch.int=Invalid number entered
typeMismatch=Invalid data entered
required=Missing field
```

messages.properties

```
# inne komunikaty
message.welcome=Welcome to our application!
```

Internacjonalizacja (5)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład użycia w kodzie

Test.java

```
public static void main(String[] args) {
  MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
  String message = resources.getMessage("message.welcome", null, "Welcome", null);
  System.out.println(message);

  String error = resources.getMessage("error.critical.required",
    new Object[] {"Config path"}, null); //exception thrown if message not found
  System.out.println(error);
}
```

Wynik działania programu

```
Welcome to our application!
The 'Config path' property is required.
```

Internacjonalizacja (6)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Dla każdej obsługiwanej wersji językowej należy przygotować osobne pliki .properties, dodając do ich nazwy identyfikator lokalizacji, np.:

messages_pl.properties

```
message.welcome=Witaj w naszej aplikacji!
```

messages_en_GB.properties

```
message.welcome=Welcome to our application, Sir!
```

messages_en_US.properties

```
message.welcome=Welcome to our application, dude!
```

Internacjonalizacja (7)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Lokalizacja (wersja językowa) może zostać ustawiona w środowisku w taki sam sposób jak w typowych aplikacjach Java np. programowo poprzez Locale.setDefault() lub poprzez przełączniki dla maszyny wirtualnej
- W metodach MessageSource można też jawnie wskazać lokalizację, podając obiekt Locale jako parametr
- W aplikacjach internetowych Spring MVC lub Spring WebFlow wersja językowa jest domyślnie pobierana z żądania od klienta i odpowiada zazwyczaj lokalizacji ustawionej w jego przeglądarce

Internacjonalizacja (8)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Jako alternatywę dla ResourceBundleMessageSource można zastosować też klasę ReloadableResourceBundleMessageSource, którą wzbogacono o:
  - możliwość wczytywania plików .properties z dowolnego miejsca w systemie plików, a nie tylko z classpath
  - możliwość wielokrotnego ładowania plików z komunikatami w trakcie działania aplikacji

Zdarzenia

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Obsługa zdarzeń w Spring oparta jest o implementacje interfejsu ApplicationListener oraz klasę ApplicationEvent zgodnie z typowym wzorcem projektowym Observer
- Wszystkie beany w kontenerze, które implementują interfejs ApplicationListener, są automatycznie powiadamiane o zdarzeniach w Spring

```
public interface ApplicationListener<E extends ApplicationEvent>
  extends EventListener {

  void onApplicationEvent(E event)
}
```

Zdarzenia (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Zdarzenia wbudowane:
  - ContextRefreshedEvent - zdarzenie zachodzi po wczytaniu konfiguracji do ApplicationContext, przy uruchamianiu lub po użyciu metody refresh() z ConfigurableApplicationContext
  - ContextStartedEvent - zdarzenie zachodzi po wykonaniu metody start() z ConfigurableApplicationContext (Lifecycle)
  - ContextStoppedEvent - zdarzenie zachodzi po wykonaniu metody stop() z ConfigurableApplicationContext
  - ContextClosedEvent - zdarzenie zachodzi po wykonaniu metody close() z ConfigurableApplicationContext
  - RequestHandledEvent - zdarzenie ma miejsce po obsłużeniu żądania HTTP, dotyczy tylko aplikacji używających DispatcherServlet

Zdarzenia (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Aby zdefiniować własny rodzaj zdarzenia należy rozszerzyć klasę ApplicationEvent, np.:

```
public class PoprawnaAutoryzacjaEvent extends ApplicationEvent {
  private String login;
  private String url;

  public PoprawnaAutoryzacjaEvent(Object source, String login, String url) {
    super(source);
    this.login = login;
    this.url = url;
  }

  //gettery, settery, pozostale metody
}
```

Zdarzenia (4)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Aby wywołać wystąpienie zdarzenia należy wywołać metodę publishEvent() z interfejsu ApplicationEventPublisher
- Aby pobrać implementację tego interfejsu w beanie można posłużyć się interfejsem ApplicationEventPublisherAware

```
@Service
public class AutoryzacjaService implements ApplicationEventPublisherAware {
  private ApplicationEventPublisher publisher;

  // z interfejsu ApplicationEventPublisherAware
  public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {
    this.publisher = publisher;
  }

  // w tej metodzie wystepuje zdarzenie PoprawnaAutoryzacjaEvent
  public void autoryzuj(String login, String url) {
    bool ok = false;
    // dokonanie autoryzacji
    if (ok) {
      PoprawnaAutoryzacjaEvent event = new PoprawnaAutoryzacjaEvent(this, login, url);
      publisher.publishEvent(event);
    }
  }
}
```

Zdarzenia (5)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Aby przechwycić wystąpienie zdarzenia należy zaimplementować w klasie beanu obsługującego zdarzenie interfejs ApplicationListener, np.:

```
@Service
public class AutoryzacjaLogger implements
                        ApplicationListener<PoprawnaAutoryzacjaEvent> {

  @Autowired
  private AutoryzacjaDao autoryzacjaDao;

  public void onApplicationEvent(PoprawnaAutoryzacjaEvent event) {
    autoryzacjaDao.zapiszPoprawnaAutoryzacje(event.getLogin(), event.getUrl())
  }
}
```

Zdarzenia - podsumowanie

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Mechanizm propagacji zdarzeń w Spring jest prosty i skuteczny w wielu zastosowaniach, ale posiada też ograniczenia:
  - zgłaszanie zdarzeń następuje synchronicznie, metoda publishEvent blokuje do czasu obsłużenia zdarzenia przez wszystkie beany nasłuchujące (można to w razie konieczności usprawnić poprzez ApplicationEventMulticaster)
  - zdarzenia są propagowane w obrębie jednego ApplicationContext
- Jeżeli aplikacja wymaga lepszego mechanizmu komunikacji (np. pomiędzy procesami) istnieje oddzielnie rozwijany projekt Spring Integration o dużo bogatszych możliwościach

Dostęp do zasobów

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Aby pokonać ograniczenia standardowej klasy java.util.URL w Spring zaprojektowano alternatywny sposób dostępu do zasobu, poprzez zunifikowany interfejs 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();
}
```

```
public interface InputStreamSource {
  InputStream getInputStream() throws IOException;
}
```

Dostęp do zasobów (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Uniwersalny interfejs Resource posiada szereg implementacji które wiążą go z określonym typem zasobu:
  - UrlResource - klasa opakowuje java.net.URL i umożliwia dostęp do zasobów które mogą być zaprezentowane przez URL (pliki, zasoby http, zasoby ftp itd.). Jest to domyślnie używana implementacja.
  - ClassPathResource - klasa umożliwia dostęp do plików względem tzw. classpath. Aby wskazać poprzez ścieżkę do zasobu tą implementację można posłużyć się prefiksem "classpath:", np.: "classpath:/pl/dmcs/plik.txt"
  - FileSystemResource - klasa opakowuje java.io.File, umożliwia dostęp do plików
  - ServletContextResource - klasa umożliwia dostęp do plików względem katalogu głównego aplikacji internetowej

Dostęp do zasobów (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- InputStreamResource - klasa pozwala na opakowanie otwartego już strumienia InputStream
- ByteArrayResource - implementacja umożliwia wygodne opakowanie bufora danych w funkcjonalność Resource

Dostęp do zasobów (4)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Obiekty które wczytują zasoby mogą implementować interfejs ResourceLoader

```
public interface ResourceLoader {
  Resource getResource(String location);
}
```

Dostęp do zasobów (5)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Wszystkie implementacje ApplicationContext implementują również ResourceLoader, zatem w zależności od rodzaju implementacji ApplicationContext, metoda getResource() będzie generować określony typ zasobów, np.:
  - Dla ClassPathXmlApplicationContext getResource() będzie zwracać obiekty klasy ClassPathResource
  - Dla FileSystemXmlApplicationContext getResource() będzie zwracać obiekty klasy FileSystemResource
  - Dla WebXmlApplicationContext getResource() będzie zwracać obiekty klasy ServletContextResource
- Oczywiście można również wymusić określony typ zasobu identyfikując go przez specyficzny prefiks w nazwie, np.: "file:/sciezka/plik.txt" (UrlResource) lub "classpath:/sciezka/plik.txt" (ClassPathResource)

Walidacja

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Mechanizm walidacji w Spring Framework został zaprojektowany w sposób umożliwiający oddzielenie tej funkcjonalności od warstwy interfejsu użytkownika i traktowanie jej jako części logiki biznesowej
- Walidacja obiektów domenowych odbywa się poprzez implementacje interfejsu Validator
- Błędy walidacji zapisywane są w obiekcie implementującym interfejs Errors

```
public interface Validator {
  boolean supports(Class clazz);
  void validate(Object target, Errors errors);
}
```

Walidacja (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład walidacji obiektu domenowego:

Uzytkownik.java

```
public class Uzytkownik {
  private String login;
  private String haslo;

  //gettery, settery, ...
}
```

UzytkownikValidator.java

```
public class UzytkownikValidator implements Validator {

  // Walidator potrafi walidowac tylko obiekty klasy Uzytkownik
  public boolean supports(Class clazz) {
    return Uzytkownik.class.equals(clazz);
  }

  public void validate(Object obj, Errors e) {
    ValidationUtils.rejectIfEmpty(e, "login", "field.required");
    Uzytkownik u = (Uzytkownik) obj;
    if (u.getHaslo().length() < 8) {
      e.rejectValue("haslo", "password.too_short");
    }
  }
}
```

Walidacja (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przy walidacji bardzo użyteczna jest klasa ValidationUtils wyposażona w metody statyczne upraszczające sprawdzanie braków w danych:
  - invokeValidator(Validator validator, Object obj, Errors errors) - wywołanie innego walidatora
  - rejectIfEmpty(Errors errors, String field, String errorCode) - zarejestrowanie błędu gdy pole "field" jest puste (null lub "" dla String)
  - rejectIfEmptyOrWhitespace(Errors errors, String field, String errorCode) - j.w. przy czym pole nie może też zawierać samych białych znaków
- Metody ValidationUtils.reject… nie wymagają podania jawnie walidowanego obiektu, ponieważ jest on również dostępny przez metodę getTarget() w Errors

Walidacja (4)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Interfejs Errors posiada również metody przydatne przy walidacji obiektów zagnieżdżonych, np.:

```
@Component
public class OsobaValidator implements Validator {
  @Autowired private AdresValidator adresValidator;

  // bool supports(Class clazz) ...

  public void validate(Object obj, Errors e) {
    Osoba osoba = (Osoba)obj;
    // walidacja pól klasy Osoba ...
    try {
      e.pushNestedPath("adresZameldowania");
      ValidationUtils.invokeValidator(adresValidator, osoba.getAdresZameldowania(), e);
    } finally {
      e.popNestedPath();
    }
    try {
      e.pushNestedPath("adresZamieszkania");
      ValidationUtils.invokeValidator(adresValidator, osoba.getAdresZamieszkania(), e);
    } finally {
      e.popNestedPath();
    }
  }
}
```

BeanWrapper

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Podstawą funkcjonalności bindowania (mapowania pól np. z interfejsu użytkownika do pól w obiektach domenowych) jest implementacja interfejsu BeanWrapper
- Interfejs BeanWrapper posiada metody oparte o semantykę ścieżki do pola, umożliwiające:
  - sprawdzanie jakie pola zawiera dany obiekt lub obiekty w nim zagnieżdżone
  - weryfikację czy dane pole jest do odczytu/zapisu
  - pobranie/ustawienie wartości pola lub wielu na raz w danym obiekcie lub w obiektach w nim zagnieżdżonych

BeanWrapper (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Składniowo ścieżka do pól może mieć postać:
  - "nazwa" - odpowiada pojedynczemu polu w klasie z metodami getNazwa() lub isNazwa() oraz setNazwa()
  - "adres.ulica" - odpowiada polu "ulica" w zagnieżdżonym obiekcie "adres", czyli metodami getAdres().getUlica() i getAdres().setUlica()
  - "miesiac[2]" - odpowiada 3 elementowi tablicy lub kolekcji "miesiąc"
  - "miesiac[Sty]" - odpowiada elementowi mapy o nazwie "miesiąc" i kluczu "Sty"

BeanWrapper (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład użycia:

```
public class Osoba {
  private String imie;
  private String nazwisko;
  private Integer wiek;
  private Adres adres;
  private List<Osoba> dzieci;
  // gettery, settery, ...
}
```

```
public class Adres {
  private String ulica;
  private String nrDomu;
  private String nrLokalu;
  private String kod;
  private String miejscowosc;
  // gettery, settery, ...
}
```

```
BeanWrapper osoba = new BeanWrapperImpl(new Osoba());
osoba.setPropertyValue("imie", "Jan");
PropertyValue value = new PropertyValue("dzieci[0].imie", "Maciej");
osoba.setPropertyValue(value);
BeanWrapper adres = new BeanWrapperImpl(new Adres());
osoba.setPropertyValue("adres", adres.getWrappedInstance());
String miejscowosc = (String)osoba.getPropertyValue("adres.miejscowosc");

Osoba o = osoba.getWrappedInstance();
```

Konwersja

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- W Spring Framework do konwersji obiektów do reprezentacji w formie łańcucha znaków i odwrotnej używa się mechanizmu JavaBeans opartego o interfejs java.beans.PropertyEditor
- Spring zawiera implementację wielu konwerterów opartych o PropertyEditor, które służą do konwersji popularnych typów
- Aby zmienić sposób konwersji określonego typu na String można zaimplementować własny konwerter

Konwersja (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład konwertera:

```
public class OsobaEditor extends PropertyEditorSupport {
  private OsobaDao osobaDao;

  public void setAsText(String text) {
    setValue(osobaDao.znajdzPoImieniuNazwisku(text)));
  }

  public String getAsText() {
    Osoba osoba = (Osoba)getValue();
    return osoba.getImie() + " " + osoba.getNazwisko();
  }
}
```

Konwersja (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Istnieje kilka możliwości zarejestrowania klasy PropertyEditor w aplikacji:
  - Automatyczna rejestracja klas PropertyEditor dla klas znajdujących się w tym samym pakiecie (np. app.domena.Osoba i app.domena.OsobaEditor)
  - Poprzez implementację metody getPropertyDescriptors() z interfejsu BeanInfo zgodnie ze specyfikacją JavaBeans
  - Poprzez wykorzystanie post-procesora CustomEditorConfigurer

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
  <property name="customEditors">
    <map>
      <entry key="app.domena.Osoba" value="app.domena.OsobaEditor"/>
    </map>
  </property>
</bean>
...
```

Konwersja (4)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Poprzez implementację interfejsu PropertyEditorRegistrar

```
public final class MojPropertyEditorRegistrar implements PropertyEditorRegistrar {
  public void registerCustomEditors(PropertyEditorRegistry registry) {
    // nalezy tutaj generowac zawsze nowe instancje konwerterow
    registry.registerCustomEditor(Osoba.class, new OsobaEditor());
    registry.registerCustomEditor(Osoba.class, "ojciec", new OjciecEditor());
    registry.registerCustomEditor(Osoba.class, "matka", new MatkaEditor());
    // ...
  }
}
```

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
  <property name="propertyEditorRegistrars">
    <list>
      <ref bean="mojPropertyEditorRegistrar"/>
    </list>
  </property>
</bean>

<bean id="mojPropertyEditorRegistrar" class="app.domain.MojPropertyEditorRegistrar"/>
```

Konwersja (5)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Nowocześniejszą alternatywą dla mechanizmu opartego o PropertyEditor jest funkcjonalność konwersji dostępna w Spring Framework od wersji 3, opartą o interfejs Converter

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {

  T convert(S source);

}
```

- Konwertery implementujące Converter muszą być thread-safe
- Metoda convert nigdy nie będzie wywołana z pustym argumentem (null), nie jest konieczne zabezpieczanie kodu pod tym względem
- Dla niewłaściwej wartości source należy wyrzucić wyjątek IllegalArgumentException
- Metoda convert może rzucać dowolne wyjątki

Konwersja (6)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykładowy konwerter:

```
public class OsobaToStringConverter<Osoba, String> {

  String convert(Osoba source) {
    return source.getImie() + " " + source.getNazwisko();
  }

}
```

Konwersja (7)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- W przypadku gdy konieczna jest kontrola konwersji dla całej hierarchii klas można zaimplementować interfejs ConverterFactory

```
package org.springframework.core.convert.converter;

public interface ConverterFactory<S, R> {

  <T extends R> Converter<S, T> getConverter(Class<T> targetType);

}
```

Konwersja (8)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Typowy przykład użycia ConverterFactory to implementacja StringToEnumConverterFactory do konwersji obiektów typów wyliczeniowych java.lang.Enum do String

```
package org.springframework.core.convert.support;
final class StringToEnumConverterFactory implements ConverterFactory<String, Enum> {

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

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

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

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

Konwersja (9)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Dla bardziej skomplikowanych przypadków konwersji można wykorzystać funkcjonalność interfejsu GenericConverter lub ConditionalGenericConverter (dla konwersji warunkowej)

```
package org.springframework.core.convert.converter;

public interface GenericConverter {

  public Set<ConvertiblePair> getConvertibleTypes();

  Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}
```

```
package org.springframework.core.convert.converter;

public interface ConditionalGenericConverter extends GenericConverter {

  boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);

}
```

Konwersja (10)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Konwersja dokonywana jest za pośrednictwem interfejsu ConversionService

```
package org.springframework.core.convert;

public interface ConversionService {

  boolean canConvert(Class<?> sourceType, Class<?> targetType);

  <T> T convert(Object source, Class<T> targetType);

  boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);

  Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}
```

Konwersja (11)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Najczęściej klasy implementujące ConversionService implementują również ConverterRegistry

```
package org.springframework.core.convert.converter;

public interface ConverterRegistry {

  void addConverter(Converter<?, ?> converter);

  void addConverter(GenericConverter converter);

  void addConverterFactory(ConverterFactory<?, ?> converterFactory);

  void removeConvertible(Class<?> sourceType, Class<?> targetType);

}
```

Konwersja (12)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Spring Framework dostarcza kilka gotowych implementacji ConversionService
- Aby zastosować domyślną implementację można wykorzystać bean ConversionServiceFactoryBean lub FormattingConversionServiceFactoryBean

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

Konwersja (12)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Spring Framework dostarcza kilka gotowych implementacji ConversionService
- Aby zastosować domyślną implementację można wykorzystać bean ConversionServiceFactoryBean lub FormattingConversionServiceFactoryBean

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

- Jeżeli w konfiguracji kontenera nie zostanie zarejestrowany bean implementujący ConversionService, w aplikacji wykorzystywany będzie starszy mechanizm konwersji przez obiekty PropertyEditor

Konwersja (13)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład wykorzystania ConversionService w kodzie:

```
@Service
public class MyService {

  @Autowired
  public MyService(ConversionService conversionService) {
    this.conversionService = conversionService;
  }

  public void doIt() {
    this.conversionService.convert(...)
  }
}
```

Formatowanie

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Nowy mechanizm konwersji jest ogólnego przeznaczenia i posiada znacznie bardziej uniwersalny charakter niż konwersja przy pomocy obiektów PropertyEditor
- W wielu aplikacjach podstawowym zastosowaniem konwersji jest formatowanie wartości pól obiektów domenowych (daty, kwoty, itd.) na widokach interfejsu użytkownika (np. strony HTML, raporty PDF itd.) oraz przetwarzanie wartości wprowadzonych przez użytkownika (np. poprzez formularz HTML)
- Formatowanie pól jest możliwe do zrealizowania przy pomocy nowego mechanizmu konwersji, jednak nie bezpośrednio, np. nie ma prostej możliwości określenia w konwerterze lokalizacji
- Dlatego też do tego problemu przygotowano osobne rozwiązanie

Formatowanie (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Do formatowania wykorzystuje się obiekty implementujące interfejs Formatter

```
package org.springframework.format;

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;

}
```

Formatowanie (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład formatera dla java.util.Date:

```
public final class DateFormatter implements Formatter<Date> {

  public String print(Date date, Locale locale) {
    if (date == null) {
      return "";
    }
    return new SimpleDateFormat("yyyy-MM-dd", locale).format(date);
  }

  public Date parse(String formatted, Locale locale) throws ParseException {
    if (formatted.length() == 0) {
      return null;
    }
    return new SimpleDateFormat("yyyy-MM-dd", locale).parse(formatted);
  }

}
```

Formatowanie (4)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Największą innowacją związaną z nowym mechanizmem formatowania jest możliwość konfiguracji formaterów w oparciu o adnotacje w klasach domenowych

```
public class Podatek {

  @NumberFormat(style=Style.PERCENT)
  private BigDecimal procent;

  @NumberFormat(style=Style.CURRENCY)
  private BigDecimal kwota;

  @DateTimeFormat(style="S-")
  private Date data;

  @DateTimeFormat(style="SS")
  private Date dataCzas;

  @DateTimeFormat(style="-S")
  private Date czas;

}
```

Formatowanie (5)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Definiowanie własnych adnotacji dla formaterów jest możliwe poprzez implementację AnnotationFormatterFactory

```
package org.springframework.format;

public interface AnnotationFormatterFactory<A extends Annotation> {

  Set<Class<?>> getFieldTypes();

  Printer<?> getPrinter(A annotation, Class<?> fieldType);

  Parser<?> getParser(A annotation, Class<?> fieldType);

}
```

Formatowanie (6)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Aby zarejestrować klasy Formatter oraz AnnotationFormatterFactory należy odwołać się do implementacji interfejsu FormatterRegistry

```
package org.springframework.format;

public interface FormatterRegistry {

  void addFormatterForFieldType(Class<?> fieldType, Printer<?> printer,
                Parser<?> parser);

  void addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);

  void addFormatterForAnnotation(AnnotationFormatterFactory<?, ?> factory);

}
```

- Najczęściej interfejs FormatterRegistry jest implementowany przez klasę implementującą ConversionService
- Przykład gotowej implementacji to FormattingConversionService

Formatowanie (6)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład przypisania adnotacji do formatera

```
@Target({ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Waluta {
  String value() default "PLN";
}
```

```
public class WalutaAnnotationFormatterFactory
         implements AnnotationFormatterFactory<Waluta> {

  public Set<Class<?>> getFieldTypes() {
    return new HashSet<Class<?>>(asList(new Class<?>[] { BigDecimal.class }));
  }

  public Printer<BigDecimal> getPrinter(Waluta annotation, Class<?> fieldType) {
    return new WalutaFormatter(annotation.value());
  }

  public Parser<BigDecimal> getParser(Waluta annotation, Class<?> fieldType) {
    return new WalutaFormatter(annotation.value());
  }
}
```

Formatowanie (7)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład przypisania adnotacji do formatera (c. d.)

```
public final class WalutaFormatter implements Formatter<BigDecimal> {

  private String waluta;

  public WalutaFormatter(String waluta) { this.waluta = waluta; }

  public String print(BigDecimal value, Locale locale) {
    if (value == null) {
      return "";
    }
    return value.toString() + " " + waluta;
  }

  public BigDecimal parse(String formatted, Locale locale) throws ParseException {
    if (formatted.length() == 0) {
      return null;
    }
    return new BigDecimal(formatted.replaceAll(waluta,""));
  }

}
```

Formatowanie (8)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład przypisania adnotacji do formatera (c. d.)

```
public class WyplataDolary {

  @Waluta("PLN")
  private BigDecimal kwotaPoczatkowa;

  private BigDecimal kurs;

  @Waluta("USD")
  private BigDecimal kwotaKoncowa;

}
```

Formatowanie (9)

- Przykładowa konfiguracja formatowania z ustawieniami domyślnymi dla SpringMVC (domyślne formatery i wsparcie dla @NumberFormat oraz @DateTimeFormat)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  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/mvc
  http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

  <mvc:annotation-driven />

</beans>
```

Formatowanie (10)

- Aby stosować niestandardowe formatery i adnotacje należy wskazać w konfiguracji SpringMVC bean ConversionService, w którym je zarejestrowano, np.:

/app/format/MyConversionServiceFactoryBean.java

```
public class MyConversionServiceFactoryBean
  extends FormattingConversionServiceFactoryBean {

  @Override
  protected void installFormatters(FormatterRegistry registry) {

    // instalacja standardowych formatterow
    super.installFormatters(registry);

    // instalacja wlasnych formatterow
    registry.addFormatterForFieldAnnotation(new WalutaAnnotationFormatterFactory())
  }

}
```

Formatowanie (11)

- Aby stosować niestandardowe formatery i adnotacje należy wskazać w konfiguracji SpringMVC bean ConversionService, w którym je zarejestrowano, np.:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  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/mvc
  http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

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

  <bean id="conversionService"
    class="app.format.MyFormattingConversionServiceFactoryBean"/>

</beans>
```

Walidacja deklaratywna

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- W Spring 3 wprowadzono również możliwość definiowania reguł walidacji poprzez adnotacje w kodzie domenowym (wg JSR-303)

```
public class Osoba {

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

  @Min(0)
  private int wiek;

}
```

- Specyfikacja JSR-303 pozwala w bardzo elastyczny sposób definiować i implementować własne reguły walidacji, ale jej złożoność wykracza znacznie poza ramy tego wykładu
- Implementacją referencyjną specyfikacji JSR-303 jest projekt Hibernate Validator posiadający bogatą dokumentację

Walidacja deklaratywna (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Aby korzystać z mechanizmu walidacji deklaratywnej wystarczy dodać w konfiguracji definicję beanu implementującego javax.validation.ValidatorFactory (np. LocalValidatorFactoryBean)
- Do działania wymagany jest Hibernate Validator jako implementacja tej technologii

```
<beans>

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

</beans>
```

Walidacja deklaratywna (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Do zdefiniowania własnej reguły walidacji (tzw. constraint) należy zdefiniować odpowiednią adnotację, np.:

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

  String message() default "Nie dzieli sie";

  Class<?>[] groups() default {};

  Class<? extends Payload>[] payload() default {};

  int value() default 1;

}
```

Walidacja deklaratywna (4)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Następnie należy zaimplementować ConstaintValidator

```
public class PodzielnePrzezValidator
  implements ContraintValidator<PodzielnePrzez, Integer> {

  private int wartosc;

  public void initialize(PodzielnePrzez constraintAnnotation) {
    this.wartosc = constraintAnnotation.value();
  }

  public boolean isValid(Integer object, ConstraintValidatorContext constraintContext) {
    if (object == null)
      return false;
    else
      return (object % wartosc) == 0;
  }

}
```

Walidacja deklaratywna (5)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład użycia w kodzie:

```
public class RzadParzysty {

  @PodzielnePrzez(2);
  private Integer numer;

  // ...

}
```

Walidacja deklaratywna (6)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- W Spring MVC możliwe jest uruchomienie walidacji przy pomocy adnotacji @javax.validation.Valid (zarówno dla walidacji w oparciu o org.springframework.validation.Validator jak i JSR-303)
  - Błędy walidacji wg JSR-303 (ContraintViolation) są automatycznie opakowywane i dodawane jako błędy w BindingResult

```
@Controller
public class OsobaController {

  @RequestMapping("/osoba", method=RequestMethod.POST)
  public void zapisz(@Valid Osoba osoba, BindingResult result) { ... }

}
```

- W pozostałych przypadkach użycia (np. w logice biznesowej) walidację uruchamia się jawnie
  - Dla walidacji Spring poprzez Validator.validate()
  - Dla JSR-303 poprzez ValidatorFactory.getValidator().validate()

Bindowanie

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Bindowanie w Spring dokonuje się przy użyciu klasy DataBinder
- W SpringMVC bindowanie dokonuje się najczęściej z użyciem odpowiednich adnotacji i zestawu tagów dla formularzy na stronach JSP, i nie ma potrzeby jawnie używać DataBinder
- Można również w szczególnych okolicznościach zastosować bindowanie z poziomu kodu

```
Osoba osoba = new Osoba();
// dane do bindowania np. w formie mapy
Map<String,Object> dane = new HashMap<String, Object>();
dane.put("imie", "Janek");

DataBinder binder = new DataBinder(osoba);
// mozna ustawic tez walidator
binder.setValidator(new OsobaValidator());
// bindowanie danych reprezentowanych poprzez interfejs PropertyValues
binder.bind(new MutablePropertyValues(dane));
// walidacja
binder.validate();
// pobranie wyniku bindowania
BindingResult results = binder.getBindingResult();
```

Spring Expression Language

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Język wyrażeń zaimplementowany w Spring jest podobny do JSP EL oraz innych implementacji
  - Umożliwia pobranie wartości pola obiektu (również zagnieżdżonego w innym obiekcie) poprzez getter
  - Wspiera indeksowanie pól kolekcji i map
  - Pozwala na wykonywanie operacji arytmetycznych
  - Umożliwia wywołanie metod obiektu oraz ustawianie wartości pól w obiektach
  - Wspiera automatyczne wyszukiwanie zmiennych w kontenerze IoC

Spring Expression Language (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- SpEL oferuje bardzo bogate możliwości i integruje się z nowym mechanizmem konwersji typów Spring
- Najczęściej używany jest do wyrażenia prostej logiki np. w konfiguracji XML

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

- Bardzo intensywnie jest też wykorzystywany w Spring Web Flow w definicjach przepływów w plikach *.flow.xml

Spring Expression Language (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Możliwe jest również jego wykonanie z poziomu kodu Java

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

Expression exp = parser.parseExpression("1 + 2 + 3");
int result = exp.getValue(Integer.class);

Osoba osoba = new Osoba();
EvaluationContext context = new StandardEvaluationContext(osoba);
Expression exp = parser.parseExpression("imie");
String imie = (String) exp.getValue(context);
exp.setValue(context, "Janek");

Expression exp = parser.parseExpression("imie = 'Jurek'");
String imie = exp.getValue(context, String.class);

System.out.println(osoba.getImie()) // Jurek
```

Spring Expression Language (4)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- W widokach JSP do wykonania wyrażeń w SpEL można użyć tagu <spring:eval> (od wersji Spring 3.0.1)
- Aby korzystać w JSP z nowego mechanizmu konwersji i formatowania Spring należy korzystać ze <spring:eval> zamiast standardowych wyrażeń EL

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

...

<spring:eval expression="1 + 2 + 3" />
```

Programowanie aspektowe

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Aspekt (aspect) - określona funkcjonalność programu, np. logowanie, zabezpieczenie dostępu, zarządzanie transakcjami
- Punkt złączenia (join point) - miejsce w kodzie programu, w którym można dokonać ingerencji np. wywołanie metody lub wyrzucenie wyjątku (w Spring AOP wykorzystywane są tylko wywołania metod)
- Rada (advice) - akcja wykonywana w momencie osiągnięcia przez wykonywany kod określonego punktu złączenia
  - Around advice - akcja otaczająca określony punkt złączenia (wywołanie metody), pozwalająca na wykonanie czynności przed i po lub zamiast metody
  - Before advice - akcja wykonywana przed wykonaniem metody, bez możliwości zablokowania jej wykonania
  - After returning advice - akcja wykonywana po normalnym wykonaniu metody (bez wyjątku)

Programowanie aspektowe (2)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- After throwing advice - akcja wykonywana po zakończeniu metody wyrzuceniem wyjątku
- After advice - akcja wykonywana po zakończeniu metody, niezależnie od tego w jaki sposób się zakończyła
- Before advice - akcja wykonywana przed wykonaniem metody, bez możliwości zablokowania jej wykonania
- After returning advice - akcja wykonywana po normalnym wykonaniu metody (bez wyjątku)

- Punkt cięcia (pointcut) - warunek dopasowania punktu złączenia do wykonania określonej rady, np. wszystkie metody o określonej nazwie
- Wstawianie (introduction) - możliwość rozszerzania klas obiektów o dodatkowe interfejsy i metody

Programowanie aspektowe (3)

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Splatanie (weaving) - mechanizm wstawiania w kodzie programu kodu aspektowego
  - Compile time weaving - splatanie w trakcie kompilacji kodu (np. przez kompilator AspectJ)
  - Load time weaving - splatanie kodu klas programu z kodem aspektowym w momencie ich ładowania przez ClassLoader
  - Runtime weaving - splatanie kodu w trakcie wykonywania programu, wykorzystywane w Spring
- Wstawianie (introduction) - możliwość rozszerzania klas obiektów o dodatkowe interfejsy i metody
- AOP proxy - obiekt opakowany w typ rozszerzony wspierający zdefiniowane aspekty

AOP Proxy

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Domyślnie Spring AOP wykorzystuje J2SE dynamic proxies, które nadają się jedynie do rozszerzania interfejsów
- Dla otaczania klas konieczne jest zastosowanie biblioteki CGLIB. Jeżeli obiekt otaczany przez proxy nie implementuje żadnego interfejsu lub konieczne jest otoczenie metody nie zadeklarowanej w interfejsie, zostanie użyte proxy CGLIB
- Dlatego nadal zalecane jest programowanie do interfejsów

Styl deklaracji @AspectJ

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- W Spring od wersji 2 można definiować aspekty z wykorzystaniem deklaracji @AspectJ, używaną w bibliotece AspectJ
- Jednak pomimo stosowania tego stylu, Spring nadal wewnętrznie generuje obiekty proxy i nie wykorzystuje pełnej funkcjonalności oferowanej przez AspectJ (np. kompilatora lub tkacza)
- Aby można było używać w Spring AOP stylu @AspectJ należy dodać do konfiguracji poniższy fragment oraz dołączyć biblioteki aspectjweaver.jar i aspectjrt.jar w wersji 1.5.1 lub nowszej

```
<aop:aspectj-autoproxy/>
```

Deklarowanie aspektu przez @AspectJ

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład definiowania aspektu przy użyciu stylu @ApectJ

```
package example;

@Component
@Aspect
public class PrzykladowyAspekt {

  // Punkt cięcia określający wszystkie metody o nazwie transfer
  @Pointcut("execution(* transfer(..))")// the pointcut expression
  private void anyTransfer() {}// the pointcut signature

  @Before("example.PrzykladowyAspekt.anyTransfer()")
  public void sprawdzDostep() { /* ... */ }

  @AfterReturning(pointcut="execution(* transfer(..))",returning="wynik")
  public void posprzataj(Object wynik) { /* ... */ }

  @Around("example.PrzykladowyAspekt.anyTransfer()")
  public Object loguj(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("przed");
    Object ret = pjp.proceed();
    System.out.println("po");
    return ret;
  }
}
```

Deklarowanie aspektu w pliku XML

dr inż. Jakub Chłapiński, jchlapi@dmcs.pl

- Przykład definiowania aspektu przy użyciu przestrzeni nazw aop

```
<aop:config>

  <aop:aspect id="mojAspekt" ref="mojAspektImpl">

    <aop:pointcut id="metodaSerwisowa"
      expression="execution(* example.service.*.*(..))"/>

    <aop:before pointcut-ref="metodaSerwisowa" method="loguj"/>

  </aop:aspect>

</aop:config>

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