Spring

Krzysztof Ślusarski

Wydział Matematyki Informatyki i Mechaniki Uniwersytetu Warszawskiego

18 maja 2007

Zalety

- Nie narzuca nam określonego modelu programowania (jak np. EJB);
- Jest lekki, nie potrzebny jest serwer aplikacji;
- Łatwosć testowania;
- Łatwość wstawiania mocków;
- Łatwość rozbudowy;
- Możliwość zmiany implementacji poszczególnych komponetów (np. DAO) bez ingerencji w inne komponenty;
- Klasy dające łatwą możliwość integracji z innymi technologiami (Hibernate, JDBC, JPA ...);
- B. dobra dokumentacja;
- Duża społeczność używająca Springa;
- Pozwala programiście skupić się na implementacji tego co naprawde ważne;

Komponenty

1. Core - zawiera implementację kontenera IoC i Dependency Incection. Najważniejszy jest tu interfejs BeanFactory.
2. DAO - zawiera narzędzia pomocne przy tworzeniu DAO za pomocą JDBC (np. nie trzeba parsowac błędów). Zawiera także mechanizm zarządzania transakcjami.
3. ORM - zawiera narzędzia pomocne przy tworzeniu DAO za pomocą popularnych ORM (np. Hibernate, JPA, JDO, Toplink, ...).
4. AOP - zawiera narzędzia dla programowania aspektowego.
5. JEE - zawiera narzędzia pomocne przy integrowaniu aplikacji z Java Enterprise Edition (JMS, JMX, EJB, Mail, ...).
6. Web - zawiera narzędzia pomocne przy tworzeniu serwisów WWW (w tym kontroler MVC).

Podstawy

Podstawowym bytem w Springu są beany, które żyją wewnątrz kontenera IoC. Beanem w sensie springowym może być każda klasa zgodna ze specyfikacja JavaBean. Beany nie muszą implementowć żadnego interfejsu, ani rozszerzać żadnej klasy. Przykładowy bean:

Podstawy ...

```
public class Bean {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Podstawy ...

Beany można deklarować na klika sposobów, my zajmiemy się tylko deklarowaniem ich w XML-u. Dla nie lubiących XML-a są propertiesy.

Podstawy ...

```
<beans>
    <import resource=''context-dao.xml'' />
    <import resource=''context-common.xml'' />

    <bean id=''...'' class=''...'' />
    <bean id=''...'' class=''...'' />
</beans>
```

Atrybuty beanów

Najważniejsze atrybuty beanów deklarowane w XMLu:

- id - identyfikator beana
- class - klasa beana
- abstract - czy jest to bean abstrakcyjny
- autoWire - autoinjectowanie
- singleton - czy bean jest singletonem
- parent - nadklasa beana
- init-method - metoda wywoływana przy inicjacji
- lazy-init - czy inicjacja ma być leniwa
- destroy-method - metoda wywoływana przy niszczeniu beana

Pobieranie beanów

Pobieranie beanów:

- fabryka beanów
- kontekst aplikacji

```
Resource resource = new ClassPathResource(''beans.xml'');
BeanFactory beanFactory = new XmlBeanFactory(resource);

ApplicationContext applicationContext = new
ClassPathXmlApplicationContext(''beans.xml'');
```

Metody fabryki beanów

- `boolean containsBean(String);`
- `Object getBean(String);`
- `Object getBean(String, Class);`
- `Class getType(String);`
- `boolean isSingleton(String);`
- `String[] getAliases(String);`

Singleton

Szczególną uwagę należy zachować przy ustawianiu atrybutu singleton. Domyślnie jest to wartość true, co oznacza, że w aplikacji będzie istnieć 1 kopia klasy (chyba, że sami utworzymy drugą za pomocą new). Jeżeli zmienimy wartość tą na true to po utworzeniu beana i przekazaniu jego referencji spring framework zapomina o nim.

Konwencje nazwenictwa

Stosuje się standardowe nazewnictwo dla Javy + JavaBeans. Nazwy beanów springowych powinny być nazwą klasy/interfejsu z pierwszą literą/wyrazem pisanym małymi literami.

Aliasy

Można stworzyć alias:

```
<alias name=''fromName'' alias=''toName'' />
```

Konwencje pisania kodu

```
public interface Dao { ... }
public class MockDaoImpl implements Dao { ... }
public class DaoImpl implements Dao { ... }
public class Clazz {
    private Dao dao;
    public setDao(Dao dao) {
        this.dao = dao;
    }
}

<bean id=''dao'' class=''MockDaoImpl'' />
<bean id=''clazz' class=''Clazz''>
    <property name=''dao'' ref=''dao'' />
</bean>
```

Inicjacja

1. Przy pomocy konstruktora:

```
<bean class=''...'' id=''...'' />
```

2. Przy pomocy statycznej metody:

```
<bean class=''...'' id=''...''
factory-method=''...'' />
```

3. Przy pomocy statycznej metody z innego beana:

```
<bean class=''...'' id=''...''
factory-method=''...'' factory-bean=''...'' />
```

Dependency Injection

1. Przy pomocy settera:

```
private Clazz clazz;

public setClazz(Clazz clazz) {
    this.clazz = clazz;
}
```

2. Przy pomocy konstruktora:

```
private Clazz clazz;

public ActClazz(Clazz clazz) {
    this.clazz = clazz;
}
```

Settery

```
<bean id=''clazz'' ... />
<bean id=''...'' class=''...''>
    <property name=''address'' value=''www.wp.pl'' />
    <property name=''port''>
        <value>102</value>
    </property>
    <property name=''clazz'' ref=''clazz'' />
    <property name=''clazz-2''>
        <ref bean=''clazz'' />
    </property>
    <property name=''clazz-3''>
        <ref local=''clazz'' />
    </property>
</bean>
```

Konstruktory

```
<bean id=''...'' class=''...''>
    <constructor-arg index=''0'' value=''102'' />
    <constructor-arg index=''1'' ref=''clazz'' />
    ...
</bean>
```

Co można przekazać jako property

- `<idref bean=''...'' />`
- `<null />`
- `<list />`
- `<set />`
- `<map />`
- `<props />`

Kolekcje i propertiesy można ze sobą merge-ować.

Listy i Sety

```
<list>
    <value>test</value>
    <ref bean=''...'' />
</list>

<set>
    <value>test</value>
    <ref bean=''...'' />
</set>
```

Mapy

```
<map>
    <entry>
        <key>
            <value>name</value>
        </key>
        <value>val</value>
    </entry>
    <entry>
        <key><value>name-2</value></key>
        <ref bean=''clazz'' />
    </entry>
    <entry key=''name-3'' value=''1'' />
</map>
```

Propertiesy

```
<props>
    <prop key=''key-1''>value</prop>
    <prop key=''key-2''>value</prop>
</props>
```

Leniwa inicjacja

Domyślnie Spring inicjuje wszystkie singletony i injectuje wszystkie propertiesy. Można ustawić, żeby bean był inicjowany dopiero w momencie odwołania się do niego.

```
<bean lazy-init=''true'' ... />
```

Wówczas część błędów nie może zostać wykryta w czasie uruchamiania aplikacji.

```
<beans default-lazy-init=''true''>
...
</beans>
```

Autowire

Możemy sprawić żeby kotener IoC sam injectował zależności.

```
<bean autowire=''...'' />
```

- no – brak automatycznego injectowania;
- byName – injectowane są beany, dla których znajduje się setter o odpowiedniej nazwie;
- byType – injectowane są beany, dla których znajduje się setter odpowiedniego typu;
- constructor – analogicznie do byType ale dla konstruktora;
- autodetect – kontener sam wybiera albo byType albo constructor;

Autowire ...

```
<beans default-autowire=''...''>
...
</beans>
```

Można wymusić, żeby bean nie był kandydatem do autoinjectowania.

```
<bean autowire-candidate=''false'' ... />
```

Gdy chcemy sami pobrać beana

Istnieje interfejs *BeanFactoryAware*. Jeżeli jakikolwiek bean implementuje ten interfejs, to ma dostęp do fabryki beanów, która go stworzyła. Interfejs zawieta jedną metodę: *setBeanFactory*(*BeanFactory*).

Zasięgi beanów

```
<bean scope=''...'' ... />
```

Możliwe zasięgi dla beanów:

1. Singleton;
2. Prototype;
3. Request;
4. Session;
5. Global session;

Internacjonalizacja

Interfejs *MessageSources*, który jest składnikiem kontekstu aplikacji zawiera metody:

- `String getMessage(String, Object[], String, Locale);`
- `String getMessage(String, Object[], Locale);`
- `String getMessage(MessageSourceResolvable, Locale);`

Pliki dla Locale

```
<bean id=''messageSource''class=''org.springframework.
context.support.ResourceBundleMessageSource''>
    <property name=''basenames''>
        <list>
            <value>format</value>
            <value>dao</value>
        </list>
    </property>
</bean>
```

Wczytane pliki:

- format.properties
- dao.properties

Co jeszcze zawiera kontekst aplikacji

Kontekst aplikacji zawiera dodatkowo:

- Obsługę eventów;
- Obsługę w aplikacjach webowych;

Wprowadzenie

Istnieją dwa style pisania aspectów w Springu:

1. anotacje;
2. XML;

Pisanie przy pomocy anotacji możliwe jest od wersji 2.0 i tym sposobem się zajmiemy.

Wprowadzenie ...

- Aspekty – deklatowane przy pomocy anotacji @*Ascpect*;
- Join point – w Springowym AOP zawsze reprezentuje wywołanie metody;
- Pointcut – zbiór join pointów;
- Advice – kiedy akcja ma być wykonana:
  - Before advice;
  - After returning advice;
  - After throwing advice;
  - After (finally) advice;
  - Around advice;

Aspect

Najprostszy aspekt:

```
@Aspect
public class MyAspect {
}

<bean id=''myAspect'' class=''MyAspect'' />
```

Pointcut

```
@Pointcut(''execution(* transfer(..))'')
public void firstPointcut() {}

@Pointcut(''within(com.blablabla.package..*)'')
public void secondPointcut() {}

@Pointcut(''execution(* com.dao.*.*(..))'')
public void daoPointcut() {}
```

Pointcut ...

```
@Pointcut(''target(com.interfaces.SomeInterface)'')
public void interfacePointcut() {}

@Pointcut(''@target(com.annotations.SomeAnnotation)'')
public void annotationPointcut() {}

@Pointcut(''@annotation(com.annotations.SomeAnnotation)'')
public void methodAnnotationPointcut() {}
```

Pointcut ...

```
@Pointcut(''args(java.io.Serializable)'')
public void serializableArgsument()

@Pointcut(''@args(com.annotations.SomeAnnotation)'')
public void argumentWithAnnotation()
```

Dodatkowo można łączyć pointcuty przy pomocy !, &&, ||.

Before advice

```
@Before(''daoPointcut()'')
public accessCheck() {
...
}

@Before(''daoPointcut() && args(account,..)'')
public accessCheck(Account account) {
...
}
```

After returning advice

```
@AfterReturning(''daoPointcut()'')
public void doSomething() {
...
}

@AfterReturning(pointcut=''daoPointcut()
                returning=''retVal'')
public void doSomething(Object retVal) {
...
}
```

After throwing advice

```
@AfterThrowing(''daoPointcut()'')
public void exceptionFound() {
...
}

@AfterThrowing(pointcut=''daoPointcut()''
               throwing=''ex'')
public void exceptionFound(Exception ex) {
...
}
```

After (finnaly) advice, around advice

```
@After(''daoPointcut()'')
public void releaseLock() {
...
}

@Around(''daoPointcut()'')
public Object doSomething(ProceedingJoinPoint pjp)
                throws Throwable {
    Object retval = pjp.proceed();
    return retval;
}
```

Komentarz

AOP Springowe daje nam sporo podstawowej funkcjonalności. Jeżeli jest ona niewystarczająca, należy użyć np. AspectJ.

Literatura

- `http://www.springframework.org/`
- Professional Java Development with the Spring Framework