Wprowadzenie

Spring 2.5

Norbert Potocki

3 lutego 2009

Wprowadzenie

- Spring – zrąb tworzenia aplikacji JEE (ale nie tylko) w języku Java (ale nie tylko :))
- powstał jako alternatywa dla "ociężałej" technologii EJB 2
- i jako alternatywa dla Struts, które uznano za źle zaprojektowane i zaimplementowane

Główne komponenty Spring

- kontener IoC
- zrąb AOP (programowanie aspektowe)
- zrąb dostępu do danych
- zrąb obsługujący transakcyjność operacji
- zrąb MVC
- zrąb autoryzacji (Spring Security)
- zrąb do testowania (Spring Testing)

Kontenery

- założenie programowania zorientowanego obiektowo: rozbicie systemu na grupy komponentów, które można w przyszłości ponownie wykorzystać
- problem: w klasycznym sposobie programowania obiekty muszą tworzyć i zarządać swoimi zależnościami, co powoduje ich niepotrzebne "związanie" z innymi obiektami
- rozwiązanie: użycie kontenera obiektów, który działa jako fabryka i rejestr obiektów dostępnych w systemie

Bez kontenera

```
public interface ReportGenerator {
    public void generate(String[][] table);
}

public class HtmlReportGenerator implements ReportGenerator {
    public void generate(String[][] table) {
        System.out.println("Generating HTML report ...");
    }
}

public class PdfReportGenerator implements ReportGenerator {
  public void generate(String[][] table) {
      System.out.println("Generating PDF report ...");
  }
}

public class ReportService {
    private ReportGenerator reportGenerator = \textit{new PdfReportGenerator()};

    public void generateAnnualReport(int year) {
        String[][] statistics = null;
        reportGenerator.generate(statistics);
    }

    public void generateMonthlyReport(int year, int month) {
        String[][] statistics = null;
        reportGenerator.generate(statistics);
    }
```

Dodajemy kontener

```
public class Container {

  public static Container instance;
  private Map<String, Object> components;

  public Container() {
    components = new HashMap<String, Object>();
    instance = this;

    ReportGenerator reportGenerator = new PdfReportGenerator();
    components.put("reportGenerator", reportGenerator);
    ReportService reportService = new ReportService();
    components.put("reportService", reportService);
  }

  public Object getComponent(String id) {
    return components.get(id);
  }
}

public class ReportService {
  private ReportGenerator reportGenerator =
        (ReportGenerator) Container.instance.getComponent("reportGenerator");

  public void generateAnnualReport(int year) { ... }
  public void generateMonthlyReport(int year, int month) { ... }
```

Przykładowe wywołanie

```
public class Main {
    public static void main(String[] args) {
        Container container = new Container();
        ReportService reportService =
            (ReportService) container.getComponent("reportService");
        reportService.generateAnnualReport(2007);
    }
}
```

Używamy wzorca projektowego - **Service Locator**

```
public class ReportService {
    private ReportGenerator reportGenerator =
        (ReportGenerator) Container.instance.getComponent("reportGenerator");
    ...
}

public class ServiceLocator {
    private static Container container = Container.instance;
    public static ReportGenerator getReportGenerator() {
        return (ReportGenerator) container.getComponent("reportGenerator");
    }
}

public class ReportService {
    private ReportGenerator reportGenerator =
        ServiceLocator.getReportGenerator();
    public void generateAnnualReport(int year) {
        ...
    }
    public void generateMonthlyReport(int year, int month) {
        ...
    }
}
```

Używamy IoC - **Dependency Injection (DI)**

```
public class ReportService {
    private ReportGenerator reportGenerator;
    public void setReportGenerator(ReportGenerator reportGenerator) {
        this.reportGenerator = reportGenerator;
    }

    public void generateAnnualReport(int year) { ... }
    public void generateMonthlyReport(int year, int month) { ... }
}

public class Container {
    private Map<String, Object> components;
    public Container() {
        components = new HashMap<String, Object>();
    ReportGenerator reportGenerator = new PdfReportGenerator();
        components.put("reportGenerator", reportGenerator);
        ReportService reportService = new ReportService();
        reportService.setReportGenerator(reportGenerator);
        components.put("reportService", reportService);
    }

    public Object getComponent(String id) {
        return components.get(id);
    }
}
```

3 typy wstrzykiwania zależności

- poprzez użycie setter'ów
- poprzez użycie konstruktorów
- poprzez użycie interfejsu

```
public interface Injectable {
    public void inject(Map<String, Object> components);
}
```

Przykład konfiguracji

```
public class SequenceGenerator {
    private String prefix;
    private int initial;
    private int counter;

    public SequenceGenerator() {}

    public SequenceGenerator(String prefix, int initial) {
        this.prefix = prefix;
        this.initial = initial;
    }

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public void setInitial(int initial) {
        this.initial = initial;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        buffer.append(prefix);
        buffer.append(initial + counter++);
    }
}
```

Przykład konfiguracji - cd.

```
<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-2.5.xsd">

<bean name="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix">
        <value>30</value>
    </property>
    <property name="initial">
        <value>100000</value>
    </property>
</bean>

</beans>

public class Main {
    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");
        SequenceGenerator generator =
            (SequenceGenerator) context.getBean("sequenceGenerator");
        System.out.println(generator.getSequence());
        System.out.println(generator.getSequence());
    }
}
```

Możliwości kontenera Spring

- konfiguracja poprzez pliki XML
- obsługa anotacji
- sprawdzanie poprawności wstrzyknięcia:
  - wyjątek: UnsatisfiedDependencyException
  - none – brak sprawdzania
  - simple – sprawdzanie typów prostych (typy bazowe oraz kolekcje)
  - objects – sprawdzanie typów złożonych
  - all – sprawdzanie wszystkich typów

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    dependency-check="simple">
    <property name="initial" value="100000" />
    <property name="prefixGenerator" ref="datePrefixGenerator" />
</bean>
```

Możliwości kontenera Spring - cd

- konfiguracja poprzez pliki XML
- obsługa anotacji
- sprawdzanie poprawności wstrzyknięcia
- automatyczne łączenie
- dziedziczenie wartości atrybutów w konfiguracji ziaren
- zasięg tworzonych ziaren:
  - singleton – pojedyńcza instancja ziarna na kontener
  - prototype – nowe ziarno przy każdym odwołaniu
  - request – nowe ziarno dla każdego żądania HTTP (tylko w kontekście aplikacji web)
  - session – jedno ziarno dla całej sesji (tylko w kontekście aplikacji web)
  - all – sprawdzanie wszystkich typów

Możliwości kontenera Spring - cd

- konfiguracja poprzez pliki XML
- obsługa anotacji
- sprawdzanie poprawności wstrzyknięcia
- automatyczne łączenie
- dziedziczenie wartości atrybutów w konfiguracji ziaren
- zasięg tworzonych ziaren
- wsparcie dla tłumaczeń
  - automatyczne wykrywanie na podstawie danych dostarczonych przez przeglądarkę
  - tekst zapisany w formacie Java Properties (klucz-wartość)

Idea **AOP - Aspect Oriented Programming**

- metodologia uzupełniająca tradycyjny paradygmat programowania obiektowego
- OOP – dobre do modelowania logiki biznesowej, ale niewygodne przy powstających podczas kodowania problemach przelotowych (zagadnienia dotyczące wielu, często niepowiązanych, modułów w aplikacji)
- przykłady:
  - logowanie komunikatów
  - walidacja parametrów wejściowych
- różne zręby pozwalające na używanie AOP:
  - AspectJ
  - JBoss AOP
  - Spring AOP

Przykładowy problem

```
public interface ArithmeticCalculator {
    public double add(double a, double  b);
    public double sub(double a, double  b);
}

public interface UnitCalculator {
    public double kilogramToPound(double kilogram);
    public double kilometerToMile(double kilometer);
}

public class ArithmeticCalculatorImpl implements ArithmeticCalculator {
    private Log log = LogFactory.getLog(this.getClass());
    public double add(double a, double b) {
      log.info("The method add() begins with " + a + ", " + b);
      ...
      log.info("The method add() ends with " + (a + b));
    }

    public double sub(double a, double b) {
        log.info("The method add() begins with " + a + ", " + b);
        ...
        log.info("The method add() ends with " + (a - b));
    }
}

public class UnitCalculatorImpl implements UnitCalculator {
    public double kilogramToPound(double kilogram) { ... }
    public double kilometerToMile(double kilometer) { ... }
```

Rozwiązanie

Użycie wzorca projektowego **Proxy** i wykorzystanie Java Reflection:

```
public class CalculatorLoggingHandler implements InvocationHandler {
    private Log log = LogFactory.getLog(this.getClass());
    private Object target;

    public CalculatorLoggingHandler(Object target) {
      this.target = target;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
      log.info("The method " + method.getName() + "() begins with "
              + Arrays.toString(args));

      Object result = method.invoke(target, args);

      log.info("The method " + method.getName() + "() ends with " + result);
      return result;
    }
}

public class Main {
    public static void main(String[] args) {
        ArithmeticCalculator arithmeticCalculatorImpl =
            new ArithmeticCalculatorImpl();
        ArithmeticCalculator arithmeticCalculator = (ArithmeticCalculator) Proxy.newProxyInstance(
            arithmeticCalculatorImpl.getClass().getClassLoader(),
            arithmeticCalculatorImpl.getClass().getInterfaces(),
                    new CalculatorLoggingHandler(arithmeticCalculatorImpl));
```

Wskazówki (Advices) w AOP

- szablony, precyzujące punkty w których można dodać nową "otoczkę"
  - Before advice – wywoływana przed wywołaniem metody
  - After returning advice – po powrocie z metody
  - After throwing advice – gdy metoda wyrzuci wyjątek
  - Around advice – dookoła metody (pozwala decydować, czy ją wywołać, czy nie)

```
public class LoggingBeforeAdvice implements MethodBeforeAdvice {
    private Log log = LogFactory.getLog(this.getClass());
    public void before(Method method, Object[] args, Object target) throws Throwable {
        log.info("The method " + method.getName() + "() begins with " + Arrays.toString(args));
    }
}
```

Wskazówki (Advices) w AOP - cd

- szablony, precyzujące punkty w których można dodać nową "otoczkę"
- rozwiązanie od Spring 2.X - anotacje przy metodach klas proxy opisujące klasę obiektów obejmowanych

```
@Aspect
public class CalculatorLoggingAspect {
    private Log log = LogFactory.getLog(this.getClass());

    @Before("execution(* ArithmeticCalculator.add(..))")
    public void logBefore() {
        log.info("The method add() begins");
    }
}
```

Podział na 3 (albo 4) warstwy

- Model
- Kontroler
- Widok
- Usługi

Przepływ w aplikacji

- główny servlet Spring $\rightarrow$
- uchwyty mapujące (handler mappers) $\rightarrow$
- kontroler (controller) $\rightarrow$
- dopasowywacz widoków (view resolver) $\rightarrow$
- widok + model (po połączeniu) $\rightarrow$wynik

Mapowanie adresu

- mapowanie dopasowuje do adresu nazwę kontrolera, który ma obsłużyć żądanie
- dostarczone w Spring MVC klasy implementujące różne formy mapowania
- wybieramy jedną z nich definiując jej instancję w konfiguracji kontenera
- przykłady:
  - tłumaczenie ostatniego trzonu adresu URL na nazwę kontrolera
  - tłumaczenie oparte na wyrażeniach regularnych

Obsługa przez kontroler

- kontroler wykonuje akcje na modelu odpowiadające żądaniu użytkownika
- w tym celu (zazwyczaj) wprowadza się warstwę usług działającą bezpośrednio na modelu
- kontroler po przetworzeniu żądania przekazuje nazwę wybranego widoku oraz model (w postaci mapy klucz-wartość)

Dopasowanie widoku

- dopasowywacz widoków łączy przekazany przez kontroler widok z rzeczywistą nazwą widoku (pliku)
- widoki mogą być zaimplementowane w np. JSP, JSF, Facelets, Tapestry
- Spring dostarcza różne dopasowywacze widoków bazujące (bądź ignorujące) przekazaną przez kontroler nazwę
- widok jest wypełniany polami przekazanymi wraz z modelem przez kontroler i przedstawiany użytkownikowi

Integracja z Spring MVC z innymi zrębami

- Direct Web Remoting
- Apache Struts
- JavaServer Faces
  - konfiguracja wymaga tylko 1 linijki kodu
  - możliwość odwoływania się do ziaren Spring w JSF i na odwrót
  - trywialna integracja ze Spring Web Flow
  - Around advice – dookoła metody (pozwala decydować, czy ją wywołać, czy nie)

Integracja Spring MVC z innymi zrębami

- Spring Security
  - implementacja najważnejszych metod autoryzacji i autentykacji
  - dostęp do zasobów oparty o role
  - listy kontroli dostępu (ACL) na poziomie pojedynczych obiektów
- Spring Web Flow
  - idea podobna do JBPM (Java Business Process Management) ale odnosząca się do interfejsu aplikacji webowych
  - łatwa integracja przepływów z akcjami JSF
  - walidatory do większości typów Javy
  - konwertery wartości
  - łatwe zarządzanie sesjami
  - łatwa integracja przepływów z Ajax
  - łatwa integracja ze Spring Security

Przydatne narzędzia

- NetBeans 6.5
  - natywne, ograniczone wsparcie dla Spring
  - brak wsparcia dla Spring Web Flow
  - automatyczne uzupełnianie kodu i znaczników XML
- Eclipse + Spring IDE (wtyczka)
  - wsparcie dla wszystkich projektów Spring
  - diagramy zależności ziaren
  - uwzględnianie ziaren tworzonych w konfiguracji kontenera Spring
  - diagramy dla Spring Web Flow
  - rozwijany równocześnie ze zrębem Spring

Przydatne materiały

- Spring in Action — Craig Walls and Ryan Breidenbach
- Pro Spring 2.5 — Jan Machacek, Jessica Ditt, Aleksa Vukotic, and Anirvan Chakraborty
- Professional Java Development with the Spring Framework — Rod Johnson, Juergen Hoeller, Alef Arendsen, and Thomas Risberg
- Pro Java™ EE Spring Patterns: Best Practices and Design Strategies Implementing Java EE Patterns with the Spring Framework — Dhrubojyoti Kayal
- http://www.springsource.org/ — główna strona projektu
- http://www.springframework.net/ — Spring Framework .NET