Das Spring Framework: eine Einführung

1. Überblick & Geschichte
2. Grundlegende Konzepte
    2.1 Module des Frameworks
    2.2 POJOs
    2.3 Dependency Injection
        2.3.1 Beispiel
        2.3.2 verschiedene Typen von DI
        2.3.3 Konfiguration mit Spring
        2.3.4 andere DI Container
    2.4 AOP
    2.5 Templates
3. Beispiele
    3.1 Unit Tests
    3.2 JMS Integration
    3.3 Transparentes Caching mit AOP
4. Quellen

1. Was ist Spring?

- OpenSource Applikationsframework für die Java-Plattform (insbesondere für J2EE)

- entstanden aus Frustration über die (oft unnötige) Komplexität bestehender J2EE Architekturen (mit EJBs und Application-Servern)

- stellt leichtgewichtige Alternative zu diesen dar

Vorteile beim Einsatz von Spring (laut den Entwicklern):

- Spring can effectively organize your middle tier objects.
- Spring can eliminate the proliferation of Singletons seen on many projects.
- Spring eliminates the need to use a variety of custom properties file formats.
- Spring facilitates good programming practice by reducing the cost of programming to interfaces, rather than classes, almost to zero.
- Spring is designed so that applications built with it depend on as few of its APIs as possible.
- Applications built using Spring are very easy to test.
- Spring helps you solve problems with the most lightweight possible infrastructure.
- Spring provides a consistent, simple programming model in many areas, making it ideal architectural "glue."

1. Geschichte

Oktober 2002

Grundlagen und Prinzipien:
Rod Johnson: Expert One-on-One J2EE Design and Development

Juni 2003

Erster Release

Oktober 2006

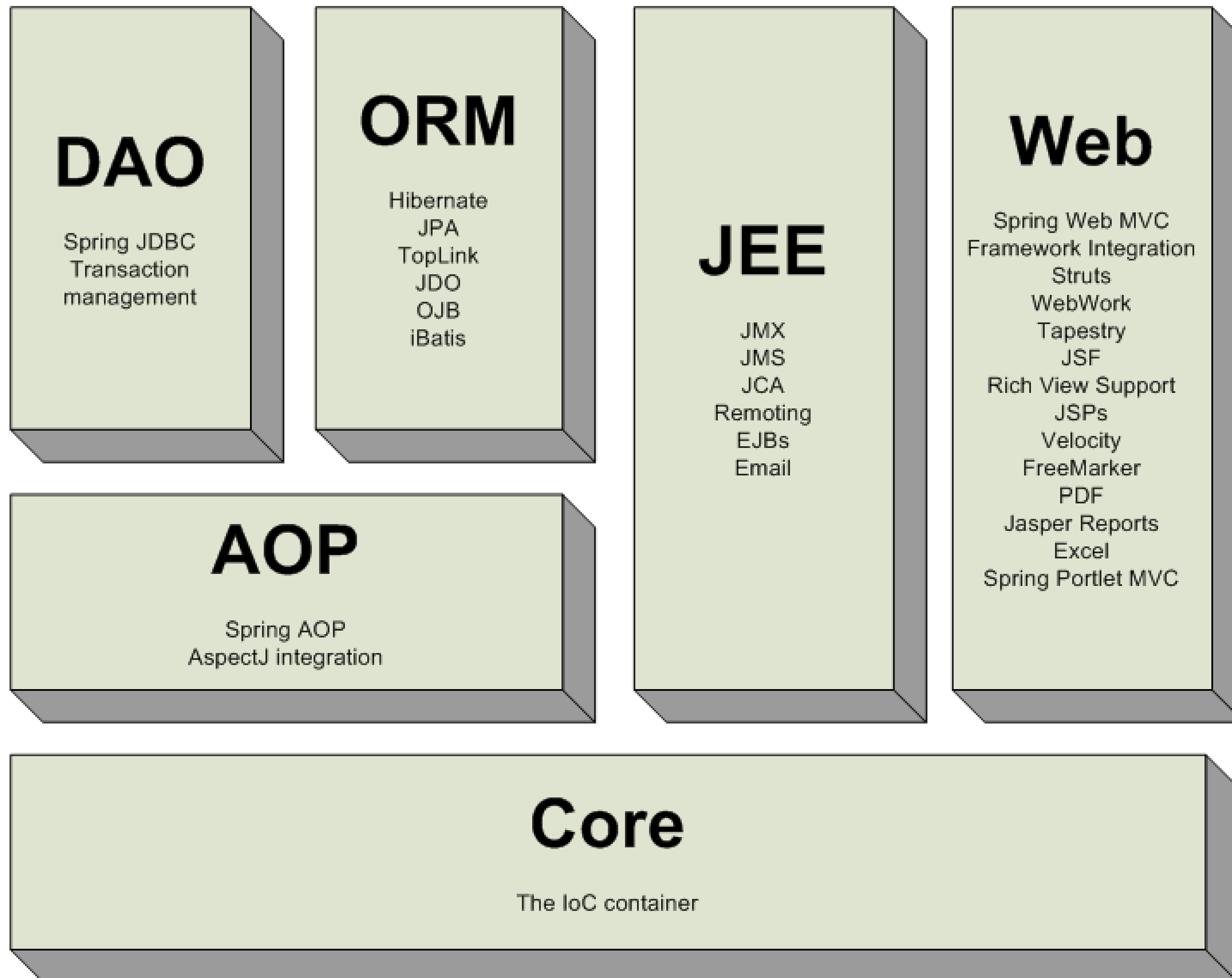
Release von Spring 2.0
(1 Million Downloads bis dahin, JAX Innovation Award 2006)

November 2007

Release von Spring 2.5

1. Architektur einer typischen Spring-basierten Webanwendung

2.1 Module des Spring Framework



2.2 POJOs

POJO: Plain Old Java Object, „ganz normales Java Objekt“

- erweitert keine vorspezifierte Klasse
- implementiert kein vorspezifiertes Interface
- muss keine vorspezifizierte Annotation enthalten

- radikaler Gegenentwurf zu EJBs (aus der Version 2.1)

- wird mit Spring lauffähig in beliebiger Umgebung: JEE Container, Servlet Container, Unit Tests, standalone...

2.3 Dependency Injection

Beispiel: Anwendung, die Filme eines bestimmten Regisseurs aus einer Datenbank liest

2 Komponenten: MovieFinder (Lesen aller vorhandenen Filme),
MovieLister (Auswahl der Filme)

```
public class Movie implements Serializable {
     private String name;
     private Director director;
     ...
}

public interface MovieFinder {
     public Movie[] findMovies();
}

public class MovieFinderImpl implements MovieFinder {
     public Movie[] findMovies() {
          // Lesen der Filme aus einer Datenbank oder von einem Webservice
     }
}
```

Wie bekommt der MovieLister nun die Implementierung des MovieFinders?

2.3.1 Naive Implementierung

```
public class MovieLister {

    public List<Movie> listMovies( String directorName ) {

        List<Movie> movies = new LinkedList<Movie>();
        MovieFinder movieFinder = new MovieFinderImpl();

        for( Movie movie : movieFinder.findMovies() ) {
            if ( movie.getDirector().getName().equals( directorName ) ) {
                movies.add( movie );
            }
        }

        return movies;
    }
}
```

- direkte Kopplung im Code
- schwer bis gar nicht testbar (da Implementierung nicht austauschbar)

2.3.1 Klassische J2EE Lösung: Service Locator

```
public class ServiceLocator {
    ...
    public Object getService( String serviceName ) {
        if ( "movieFinder".equals( serviceName ) ) {
            // Holen per JNDI
        }
        ...
    }
}

public class MovieLister {
    public List<Movie> listMovies( String directorName ) {
        ServiceLocator locator = ServiceLocator.getInstance();
        MovieFinder finder = (MovieFinder)locator.getService("movieFinder");
    ...
}
```

- Testbarkeit abhängig von der Implementierung des Service Locators
- zusätzliche Klasse notwendig
- aufrufende Komponente muss Service Locator kennen

2.3.1 Dependency Injection

Hollywood-Prinzip: „Don't call us, we call you“

- Klasse bekommt benötigte Komponenten von außen „injiziert“
- Klasse muss nur wissen was sie braucht, nicht wie sie es bekommt
- DI Container garantiert, dass Abhängigkeiten gesetzt werden
- entkoppelt Klasse von der Laufzeitumgebung

```
public class MovieLister {

    private MovieFinder movieFinder;
    ...
}
```

Übergabe durch den Konstruktor (Constructor Injection)

```
public MovieLister( MovieFinder finder ) { this.movieFinder = finder; }
```

Übergabe durch Setter-Methode (Setter Injection)

```
public void setMovieFinder( MovieFinder finder ) { this.movieFinder = finder; }
```

2.3.2 Vor- und Nachteile verschiedener DI-Typen

Constructor Injection

+ Klasse immer in gültigen Zustand

- nicht kompatibel zum JavaBeans Standard
- zuviele Argumente bei vielen Abhängigkeiten
- Argumente müssen in Subklassen mitgenommen werden
- für Tests müssen alle Abhängigkeiten gesetzt werden

Setter Injection

+ kompatibel zum JavaBeans Standard
+ für Tests können einzelne Abhängigkeiten gesetzt werden

- Klasse kann in ungültigem Zustand sein
- viel Code notwendig

2.3.3 Konfiguration mit Spring

MovieFinder:

```
<bean id="movieFinder" class="de.fuberlin.spring.MovieFinderImpl"/>
```

MovieLister mit Setter Injection:

```
<bean id="movieLister" class="de.fuberlin.spring.MovieLister">
    <property name="movieFinder" ref="movieFinder"/>
</bean>
```

MovieLister mit Constructor Injection:

```
<bean id="movieLister" class="de.fuberlin.spring.MovieLister">
    <constructor-arg ref="movieFinder"/>
</bean>
```

- *„ApplicationContext“* wird zur Laufzeit initialisiert
- Komponenten (*„Beans“*) werden erzeugt und verknüpft
- *„Beans“* werden als Singletons erzeugt, ohne explizit den Pattern implementieren zu müssen

2.3.3 Konfigurationsmöglichkeiten

XML Konfiguration

Komplexe Datenstrukturen können abgebildet werden (Properties, Lists, Maps, Enums...)

```
<bean id="moreComplexObject" class="example.ComplexObject">
  <property name="adminEmails">
    <props>
        <prop key="administrator">administrator@example.org</prop>
        <prop key="support">support@example.org</prop>
    </props>
  </property>
  <property name="someMap">
    <map>
        <entry>
            <key><value>a ref</value></key>
            <ref bean="myDataSource" />
        </entry>
    </map>
  </property>
</bean>
```

2.3.3 Konfigurationsmöglichkeiten

Konfiguration über Annotations

bietet nur einen Teil der Möglichkeiten der XML-Konfiguration

@Component / @Service / @Repository
markiert Klasse als Spring-Bean

@Autowired
verlangt Injection

@Qualifier
trifft Auswahl bei mehreren Injection-Kandidaten

```
@Service
public class MovieLister {

    @Autowired
    @Qualifier("movieFinder")
    private MovieFinder movieFinder;
}
```

2.3.4 Weitere DI Container

PicoContainer
Konfiguration via Java Code

Google Guice
Konfiguration via Java Code und Annotations

Hivemind
Konfiguration via XML

(Im Prinzip alle gleichmächtig, Entscheidung eher Frage des persönlichen Geschmacks)

2.4 AOP

Beispiel: Transaktionalität

Anwendung macht Datenbankzugriffe in der Persistenzschicht, es muss nun in der Anwendungslogik festgelegt werden, welche Zugriffe jeweils in einer Transaktion ablaufen

Sollte an einer Stelle konfigurierbar sein, nicht an zig Stellen codiert

Wie kann man das technisch realisieren?

„Cross Cutting Concern“, querschnittlicher Belang

Anforderung an eine Anwendung, die nicht in einer einzelnen Schicht modular behandelt werden kann (z.B. Transaktionalität, Logging, Caching, Sicherheit)

Aspektorientierte Programmierung

ermöglicht es diese Belange gekapselt zu modellieren und deklarativ modulübergreifend verschiedenen Codefragmenten zuzuordnen

2.4 Deklaratives Transaktions-management mit Spring AOP

Spring AOP: nur auf Methodenaufrufe anwendbar, Beans werden im Container mit Proxies versehen, die die erweiterte Funktionalität aufrufen (ähnlich dem Decorator-Pattern)

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes><tx:method name="listMovies" read-only="false" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>

<aop:config>
  <aop:pointcut id="txM" expression="execution(* de.fuberlin.MovieLister.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="txM"/>
</aop:config>

public class MovieLister {
     ...
     @Transactional
     public List<Movie> listMovies( String directorName ) { ... }
}
```

Per AOP wird vor dem Methodenaufruf eine Transaktion gestartet, nach dem Aufruf wird diese automatisch commitet oder zurückgerollt

2.5 Templates

Hilfsklassen für verschiedene APIs, die automatisches Resourcenmanagement, einheitliche Fehlerbehandlung, Vereinfachung der API und diverse Hilfsmethoden bieten

- JmsTemplate
- JdbcTemplate
- HibernateTemplate
- ...

JdbcTemplate

```
queryForObject( String sql, Object[] args, Class requiredType );
```

JmsTemplate

```
convertAndSend( String destinationName, Object message );
```

3.1 Unit Tests

```
public class MockMovieFinder implements MovieFinder {
    public Movie[] findMovies() {
        return new Movie[] {
            new Movie( "Lord of the Rings", new Director( "Peter Jackson" ) ),
            new Movie( "Batman Begins", new Director( "Christopher Nolan" ) )
        };
    }
}

@Test
public void some() {

    MovieLister movieLister = new MovieLister();
    movieLister.setMovieFinder( new MockMovieFinder() );

    List<Movie> movies = movieLister.listMovies( "Christopher Nolan" );

    assertEquals( 1, movies.size() );
    assertEquals( "Christopher Nolan", movies.get(0).getDirector().getName() );
}
```

3.3 JMS Integration: Senden

```
<amq:connectionFactory id="cFactory" brokerURL="tcp://localhost:61616"/>
<amq:queue id="movieQueue" physicalName="movieQueue"/>

<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <constructor-arg ref="cFactory" />
</bean>
```

```
@Component
public class MoviePublisher {
    ...
    @Autowired
    private JmsTemplate jmsTemplate;

    public void publish() {
        for ( Movie movie : movieLister.listMovies( "Peter Jackson" ) ) {
            jmsTemplate.convertAndSend( "movieQueue", movie );
        }
    }
}
```

- JmsTemplate kümmert sich automatisch um Resourcen (Connection etc.)
- JmsTemplate benutzt automatisch ObjectMessages

3.3 JMS Integration: Empfang

```
@Component
public class MovieConsumer {
    public void consume( Movie movie ) {
        ...
    }
}
```

wird zu „Message Driven POJO“:

```
<jms:listener-container>
    <jms:listener destination="movieQueue"
                  ref="movieConsumer" method="consume"/>
</jms:listener-container>
```

ListenerContainer von Spring holt Message aus der Queue, bestätigt diese, konvertiert das Movie Objekt zurück und ruft den MovieConsumer auf

(Adapter Pattern)

3.4 Transparentes Caching mit AOP

Annotation definieren: `public @interface Cachable {}`

Methode annotieren: `@Cachable public Movie[] findMovies() { ... }`

Spring Konfiguration anpassen:

```
<aop:aspectj-autoproxy/>
<bean id="cachingAspect" class="...CachingAspect"/>
```

```
@Aspect
public class CachingAspect {
    ...
    @Around( "@annotation( de.fuberlin.spring.Cachable )" )
    public Object cache( ProceedingJoinPoint joinPoint ) throws Throwable {
        String key = ...

        if ( !cache.containsKey( key ) ) {
            cache.put( key, joinPoint.proceed() );
        }
        return cache.get( key );
    }
}
```

Sind die Versprechen eingehalten?

- Spring can effectively organize your middle tier objects.
- Spring can eliminate the proliferation of Singletons seen on many projects.
- Spring eliminates the need to use a variety of custom properties file formats.
- Spring facilitates good programming practice by reducing the cost of programming to interfaces, rather than classes, almost to zero.
- Spring is designed so that applications built with it depend on as few of its APIs as possible.
- Applications built using Spring are very easy to test.
- Spring helps you solve problems with the most lightweight possible infrastructure.
- Spring provides a consistent, simple programming model in many areas, making it ideal architectural "glue."

4. Quellen

Spring Dokumentation
http://static.springframework.org/spring/docs/2.5.x/reference/index.html

Artikel von Rod Johnson zum Release von Spring 2.5
http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25

Martin Fowler:
Inversion of Control Containers and the Dependency Injection pattern
http://martinfowler.com/articles/injection.html

Diskussion über Dependency Injection
http://www.theserverside.com/news/thread.tss?thread_id=23358

Spring Framework bei Wikipedia
http://de.wikipedia.org/wiki/Spring_(Framework)

AOP bei Wikipedia
http://de.wikipedia.org/wiki/AOP