



Højniveau introduktion til Spring Framework



Jeppé Cramon

Kontakt: jeppe@cramon.dk



Spring's mål

- At gøre J2EE udvikling nemmere - Spring konkurrerer ikke med J2EE, men med EJB'ere og in-house applikations frameworks
- Give dig et godt og fleksibelt applikations framework uden at introducere hårde framework bindinger i din kode (et ikke invaderende framework)
- Godt OO design er vigtigere end den underliggende teknologi
- Giver dig en sammenhængende arkitektur, der samler best-of-breed single-tier arkitekturer.
- Giver dig konsistent struktur til alle arkitekturelle lag, ikke kun til f.eks. Web laget



Spring baggrund

- Tankerne bag Spring er beskrevet af Rod Johnson i bogen “Export one-on-one J2EE Design and Development”
- Spring er skrevet på baggrund af virkelige projekter og ikke design-by-committee (tm)
- Springs kode er af høj kvalitet og Unit-test dækningsgraden er stor.
- Spring har god og gratis dokumentation. Der findes et stadigt stigende antal bøger om Spring (pt. 7 bøger)



Spring baggrund

- Spring er OpenSource
- Spring er et Lightweight Framework
- Spring er en del af den Agile bevægelse
- Interface21 (folkene bag Spring) samt BEA tilbyder professionel Spring support
- Aktivt community på <http://www.springframework.org>



Fordele ved Spring

- Du kan vælge at bruge de dele af Spring der giver mening for dit projekt
- Giver dig gode service abstraktioner
- Spring implementerer low level infrastruktur kode, så du ikke selv skal skrive det
- Fjerner behovet for singletons i din kode
- Gøre det nemmere at programmere mod interfaces (Best practice)
- Gør unit testing nemmere



Fordele ved Spring

- Integrerer med andre OpenSource projekter – Ingen grund til at genopfinde den dybe tallerken
- Spring gør det muligt at adskille konfigurationen og opsætningen af applikationen fra selve applikations koden
- Spring frameworket giver mulighed for en stor grad af afkobling mellem klasser og API'er => Mindre lock-in og mere genbrug.
- Giver dig deklarative services til dine POJO's

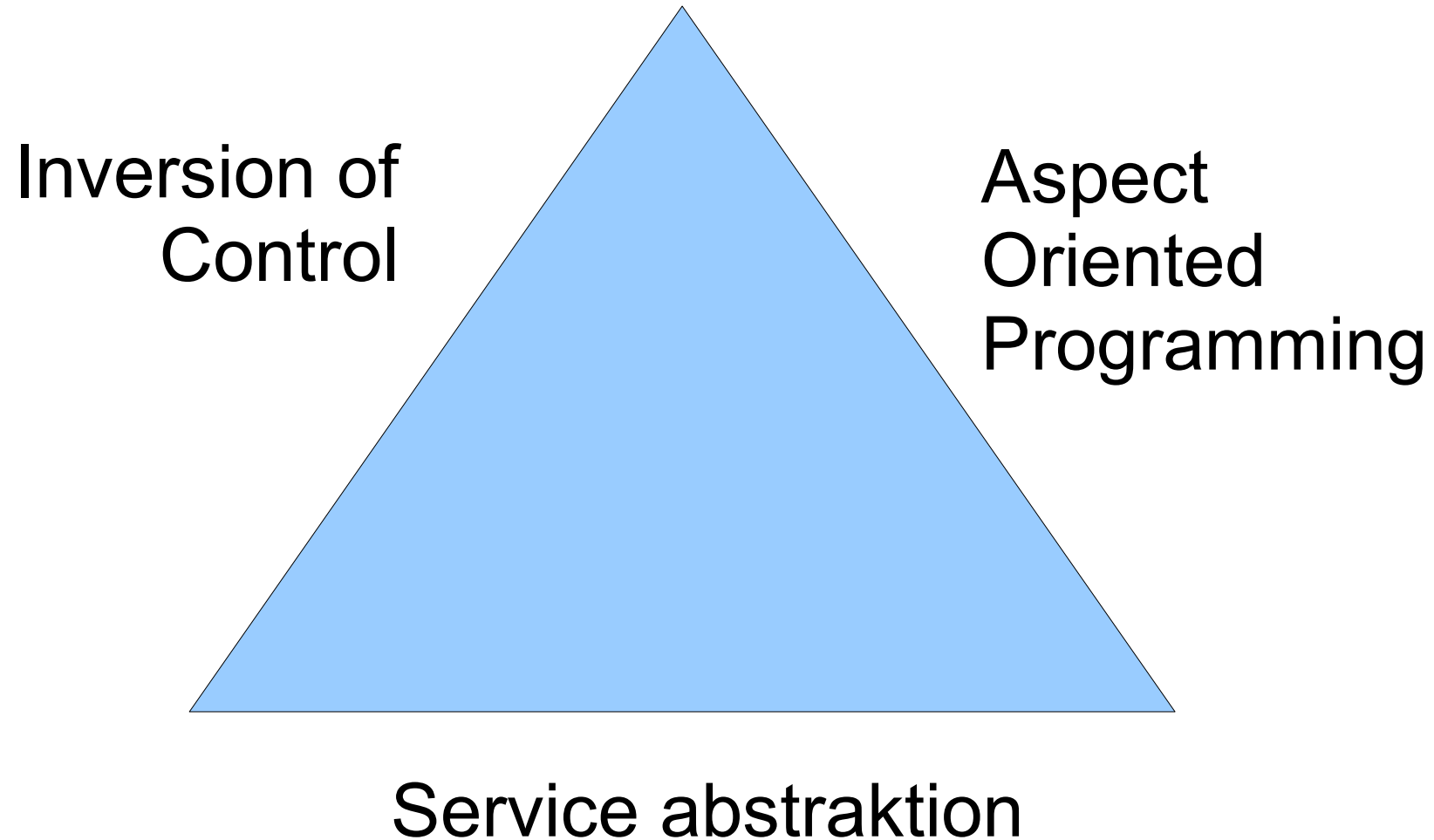


Fordele ved Spring

- Du kan nu vælge at bruge EJB'ere når det giver mening - Spring gør det muligt at udvikle Enterprise applikationer uden EJB'er eller special interfaces – *“Return of the POJO's (Plain Old Java Objects)”* ;-)
- Være kompatibel mellem forskellige applikations servere – Kan køre stand-alone i en standard Java Applikation, i en Applet, Swing eller køre under TomCat, Resin, WebLogic, WebSphere, Oracle, osv.
- Alt dette betyder, at du kan bruge mere tid på at skrive kode der giver reel merværdi for dine kunder.

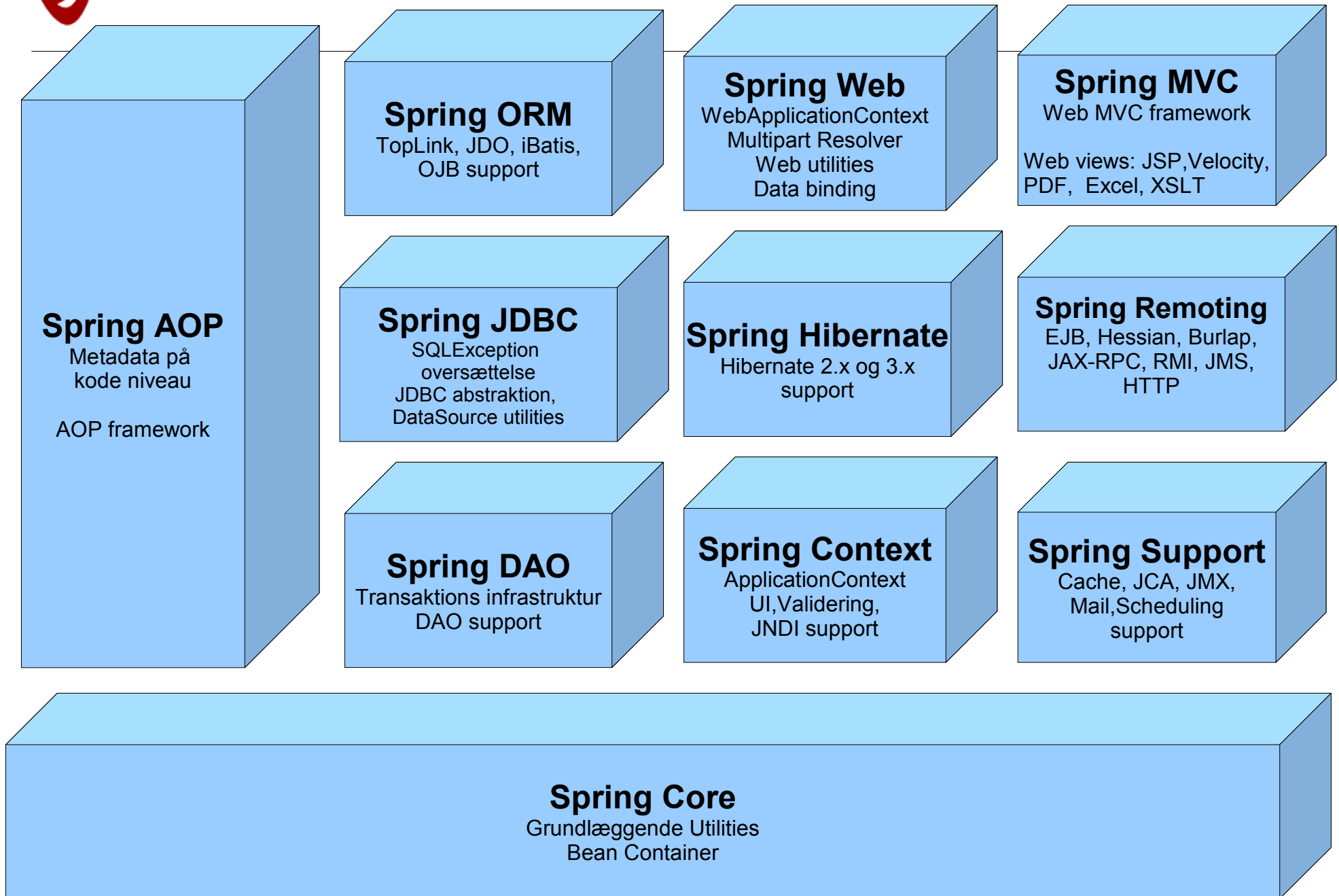


Spring – Kerne koncepter





Spring Moduler





Introduktion til Inversion of Control

- Hvad er der galt med de nuværende måder at håndtere objekt afhængigheder?
- Hvordan kan Inversion of Control hjælpe?
- Hvordan bruger man Inversion of Control sammen med Spring?



Objekt afhængigheder

Der findes to metoder hvorpå et objekts kan få referencer til andre objekter:

- Objektet kan selv hente sine afhængigheder. Dette kaldes for ***pull*** metoden. Afhængighederne er ikke synlige og er dermed **implicitte** for omverdenen
- Alternativt kan objektet, via forskellige mekanismer, specificere hvilke afhængigheder det har. Derefter, er det koden der bruger objektet, der er ansvarlig for at forsyne objektet med dets afhængigheder. Dette kaldes for ***push*** metoden. Afhængighederne er synlige og dermed **eksplicitte** for omverdenen.



Objekt referencer i Java applikationer

- Hvordan hentes referencer til andre objekter normalt i Java applikationer?
 1. Hårdt kodede referencer til implementations klasser ved brug af **new** operatoren (**pull**)
 2. Semi hårdt kodede referencer til objekter der hentes via et **Singleton Factory** (**pull**)
 3. Løse referencer til objekter der bliver hentet via **JNDI** (**pull**)
 4. Kombination af nr. 2 og 3, også kendt som **ServiceLocator** pattern (**pull**)



Objekt referencer – new operatoren

```
public class JdbcUserDao implements IUserDao {
    private DataSource dataSource;

    public JdbcUserDao() {
        super();
        dataSource = ...; // ???
    }

    public List findAll() {
        return ....;
    }
}

public class Client {
    public List findAllUsers() {
        IUserDao userDao = new JdbcUserDao();
        return userDao.findAll();
    }
}
```



Objekt referencer – `new` operatoren

Fordele:

- Ingen ekstern konfiguration
- Nemt at komme igang med

Ulemper:

- Hårdt kodede referencer. Ønsker vi at bruge en anden implementation skal vi ind og ændre alle de steder `JdbcUserDao` bliver brugt
- Gør unit testing sværere, da vi ikke nemt kan erstatte `JdbcUserDao` med et mock eller stub implementation.
- Hvordan skal `JdbcUserDao` få fat i sin `DataSource`?
Her fungerer `new` ikke.



Objekt referencer – Singleton Factory

```
public class JdbcUserDao implements IUserDao {
    private DataSource dataSource;

    public JdbcUserDao() {
        super();
        dataSource = ResourceFactory.getInstance().getDS();
    }

    public List findAll() {
        return ....;
    }
}

public class Client {
    public List findAllUsers() {
        IUserDao userDao = DaoFactory.getInstance().getUserDao();
        return userDao.findAll();
    }
}
```



Objekt referencer – Singleton Factory

Fordele:

- Ingen “direkte” hård kodning mod de implementations klasser vi skal bruge en instans af
- Vi kan nemmere udskifte implementationen af IUserDao
- Det er lidt nemmere at unitteste, da man kan konfigurere Singleton Factory'en til at returnere en mock eller stub
- Mulighed for indkapsling af afhængigheds kontrol. Vi kan lade Singleton Factory'en konfigurere JdbcDao'en med en reference til DataSourceen, påbekostning af yderligere afhængigheder i DaoFactory'en

Ulemper:

- Der er stadig hårdt kodede referencer, denne gang til et Singleton Factory. Bindingen findes i både JdbcUserDao og Client.
 - Singleton Factory'en skal have en eller anden form for konfiguration.
 - Det kan enten foregå eksternt gennem f.eks. property/xml fil
 - Eller være hårdt kodet inde i Singleton Factory'en
 - Unit testing er stadig forholdsvis kompliceret da vi skal parametrisere Singleton Factory'en for at få vores mock eller stub.
 - Hvordan håndterer vi nemt at der kan findes flere forskellige DataSources alt efter Dao uden i dette tilfælde at “forurene” ResourceFactory'en?
-



Objekt referencer - JNDI

```
public class JdbcUserDao implements IUserDao{
    private DataSource dataSource;

    public JdbcUserDao() {
        super();
        try {
            InitialContext ctx = new InitialContext();
            dataSource = (DataSource) ctx.lookup("jdbc/DS");
        } catch (NamingException e) {
            throw new DaoException(e);
        }
    }

    public List findAll() {
        return null;
    }
}

public class Client {
    public List findAllUsers() {
        IUserDao userDao;
        try {
            InitialContext ctx = new InitialContext();
            userDao = (IUserDao) ctx.lookup("mynamespace/UserDao");
        } catch (NamingException e) {
            throw new DaoException(e);
        }
        return userDao.findAll();
    }
}
```



Objekt referencer - JNDI

Fordele:

- Ingen “direkte” hård kodning mod de implementations klasser vi skal bruge en instans af
- Vi kan nemmere udskifte implementationen af IUserDao
- Vi kan nemmere håndtere flere DataSources alt efter Dao behov

Ulemper:

- Der er stadig hårdt kodede referencer, denne gang til InitialContext. Bindingen findes i både JdbcUserDao og Client.
- JNDI navnene er hårdt kodede. Hvad hvis de ændrer sig (f.eks. pga. miljø afhængigheder)
- JNDI skal konfigureres
- JNDI kan ikke håndtere afhængigheder mellem objekter for os
- Unit testing er stadig forholdsvis kompliceret da JNDI er besværligt at stubbe



Objekt referencer – ServiceLocator

```
public class ServiceLocator {
    ...
    private InitialContext ctx;
    ...
    public IUserDao getUserDao() {
        try {
            return (IUserDao) ctx.lookup("mynamespace/UserDao");
        } catch (NamingException e) {
            throw new DaoException(e);
        }
    }
}

public class JdbcUserDao implements IUserDao {
    private DataSource dataSource;

    public JdbcUserDao() {
        super();
        dataSource = ServiceLocator.getInstance().getDS();
    }

    public List findAll() {
        return ....;
    }
}

public class Client {
    public List findAllUsers() {
        IUserDao userDao = ServiceLocator.getInstance().getUserDao();
        return userDao.findAll();
    }
}
```



Objekt referencer – ServiceLocator

ServiceLocator pattern'et, er en J2EE variant af Singleton Factory kombinationen der indkapsler JNDI tilgang.

Fordele:

- Ingen “direkte” hård kodning mod de implementations klasser vi skal bruge en instans af
- Vi kan nemmere udskifte implementationen af IUserDao
- Mulighed for indkapsling af afhængigheder. Vi kan lade ServiceLocator'en konfigurere JdbcDao'en med en reference til DataSource'en.

Ulemper:

- Der er stadig hårdt kodede referencer, denne gang til ServiceLocator'en. Bindningen findes i både JdbcUserDao og Client.
- Unit testing er stadig forholdsvis kompliceret da vi skal parametrisere ServiceLocator'en for at få vores mock eller stub.
- JNDI navnene er stadig hårdt kodede. Det er bedre end ren JNDI, da de er lokaliseret til ServiceLocatoren.
- JNDI skal konfigureres
- Hvordan håndterer vi nemt at der kan findes flere forskellige DataSources alt efter Dao?



Hvad er løsningen?

1. Vi har brug for en “Singleton” Factory der kan sikre afkobling mellem interface og implementation samt for at få fat i ressourcer.
 2. Vi har brug for en “Singleton” Factory der kan håndtere afhængigheder mellem objekter, så vi undgår afhængigheder til Singleton Factory'en i alle klasser.
 3. For at understøtte dette har vi brug for en måde hvorpå klasser eksplicit kan specificere hvilke afhængigheder de har
-



Inversion of Control (IoC)

IoC, som også kaldes for Hollywood metoden (“Don't call me, I'll call you!”), følger **push** metoden.

Ansvar for hvordan referencer til andre objekter bliver opnået er vendt om (Inversion). Objektet er ikke længere ansvarlig for selv at hente referencer til andre objekter eller konfigurations parametre. Det er nu omverdenens opgave!

Væk er Factories, Singletons og `new` operatoren.

IoC er ikke kun begrænset til objekt instantiering



Inversion of Control (IoC)

- IoC hjælper dig med at opretholde interface og implementations adskillelse, da du ikke behøver at kende implementationen eller hvordan man får en reference til implementationen. Der gives en løsere kobling mellem objekterne i din applikation.
- En løsere kobling mellem objekter gør det nemmere at teste dem i isolation og genbruge dem.



Dependency Injection (DI)

- *DI* er en variant af *Inversion of Control (IoC)* princippet hvor afhængighederne (objekt referencer eller konfigurations parametre) bliver **injectet** ind i objektet.
- I stedet for at bruge f.eks. framework specifikke interfaces til at hente referencer til andre objekter eller konfigurations parametre, bruges i stedet Java sprogets konstruktioner til at specificere afhængighederne.



Dependency Injection

Typiske former for dependency injection

- Konstruktør injection
- Setter property injection
- Method injection



Dependency Injection

Konstruktør injection

Afhængigheder bliver specificeret via Konstruktør parametre

```
public class JdbcUserDao implements IUserDao {
    private DataSource dataSource;

    public JdbcUserDao(DataSource source) {
        this.dataSource = source;
    }
    ...
}
```



Dependency Injection

Setter property injection

Afhængigheder bliver specificeret via setter property

```
public class JdbcUserDao implements IUserDao {
    private DataSource dataSource;

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

    ...
}
```



Objekt referencer med DI

```
public class JdbcUserDao implements IUserDao {
    private DataSource dataSource;

    public JdbcUserDao(DataSource source) {
        this.dataSource = source;
    }

    public List findAll() {
        return ....;
    }
}

public class Client {
    public List findAllUsers() {
        DataSource dataSource = ???;
        IUserDao userDao = new JdbcUserDao(dataSource);
        return userDao.findAll();
    }
}
```



Objekt referencer med DI

Fordele:

- Ingen ekstern konfiguration
- Nemt at komme igang med
- JdbcUserDao har indeholder ingen hårdtkodede afhængigheder til implementationer eller Singleton Factories

Ulemper:

- Hårdt kodede referencer. Ønsker vi at bruge en anden implementation skal vi ind og ændre alle de steder JdbcUserDao bliver brugt
- Gør unit testing sværere, da vi ikke nemt kan erstatte JdbcUserDao med et mock eller stub implementation.
- Vi mangler stadig en måde at få fat i DataSource'en. Return of the Singleton Factory...



Spring og Dependency Injection

Spring Core indeholder en *Inversion of Control* Container kaldet **BeanFactory**, der benytter *Dependency Injection* (DI) til at styre afhængigheder og konfigurations parametre, så vi kan overkomme mange af ulemperne fra forrige slide.

Du kan konfigurere en **BeanFactory** via:

- XML fil(er) - den foretrukne måde
- Property fil(er)
- Java kode



Dependency Injection og Wiring

Spring terminologi:

Wiring: Specifikation af afhængigheder/referencer/bindinger mellem objekter

Dependency Injection og Wiring

- En Dependency Injection Container håndterer instantierings rækkefølgen mellem objekterne.
- En Dependency Injection Container kan i flere tilfælde automatisk håndtere instantieringen af objekter uden angivelse af specifikke wirings. Dette kaldes **Auto Wiring**.



Objekter afhængigheder med Spring

```
<beans>
  <bean id="userDao" class="dk.cramon.springdemo.dao.JdbcUserDao"
        autowire="constructor"/>
  <bean id="dataSource"
        class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/DS"/>
  </bean>
</beans>
```

```
public class JdbcUserDao implements IUserDao {
    private DataSource dataSource;

    public JdbcUserDao(DataSource source) {
        this.dataSource = source;
    }
    ...
}

public class Client {
    private BeanFactory beanFactory;
    public Client() {
        BeanFactory beanFactory = new XmlBeanFactory(new
            ClassPathResource("MyConfig.xml"));
    }

    public List findAllUsers() {
        UserDao userDao = (IUserDao) beanFactory.getBean("userDao");
        return userDao.findAll();
    }
}
```



Initialisering af BeanFactory

**Det er ikke meningen at applikations koden
nogensinde skal se et BeanFactory!!!**

Initialiseringen af dit BeanFactory, sker oftest igennem en Servlet, ServletListener eller main(), der ved applikations opstart indlæser konfigurationen.

Ved at lade wiringen starte ved det yderste lag, f.eks. web laget, vil frameworket automatisk håndtere initialiseringen af hele objekt hierarkiet.

Dvs. at når din Action/Controller bliver kaldt er den allerede fuldt konfigureret og du ser intet til Spring Containeren...

På den måde er den del af koden der ser BeanFactory'en begrænset til en lille smule **glue code** og intet af din applikations koden ser nogensinde et BeanFactory.



Andre Spring features

Du kan registrere to typer af beans i et BeanFactory:

Singleton (kun een instans – *standard*)

```
<bean id="userDao" class="dk.cramon.springdemo.dao.UserDao" singleton="true">
```

Prototype (en ny instans hver gang `getBean()` bliver kaldt)

```
<bean id="userDao" class="dk.cramon.springdemo.dao.UserDao" singleton="false">
```



Andre Spring features

En `FactoryBean` fungerer som et factory og defineres på samme måde som `JavaBeans`. Et kald til `getBean()` returnerer ikke `FactoryBean`'en, men i stedet det objekt factory'en er konfigureret til at returnere, f.eks. et objekt hentet fra JNDI som tilføjet med `JndiObjectFactoryBean`

```
<bean id="userDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">  
  <property name="jndiName" value="jdbc/UserSource"/>  
</bean>
```



Eksplicitte objekt afhængigheder

I dette eksempel skal UserDao og AuthorityDao skal have hver deres DataSource. Da der er to DataSources kan Spring ikke auto-wire. I stedet skal **Wirings skal angives eksplicit.**

```
<beans>
  <bean id="userService" class="dk.cramon.springdemo.service.UserService"
        autowire="constructor"/>
  <bean id="userDao" class="dk.cramon.springdemo.dao.UserDao">
    <constructor-arg ref="userDataSource"/>
  </bean>
  <bean id="authorityDao" class="dk.cramon.springdemo.dao.AuthorityDao">
    <constructor-arg ref="securityDataSource"/>
  </bean>
  <bean id="userDataSource"
        class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/UserSource"/>
  </bean>
  <bean id="securityDataSource"
        class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/SecuritySource"/>
  </bean>
</beans>
```



Spring Core – BeanFactory

Hvornår skal man bruge konstruktør vs. setter property injection til at angive afhængigheder?

Vælg konstruktør injection når:

- Du vil lave en kontrakt for din klasse. Du kan ikke instantiere klassen uden at opfylde kontrakten
- Vil sikre uforanderlighed (immutability) for nogle attributter i objektet
- Du vil undgå at bruge setter properties, blot for at kunne angive afhængigheder

Vælg setter property injection når:

- Du vil undgå for lange konstruktører signaturer
- Gør det nemmere at håndtere default eller optionelle værdier
- Du vil undgå problemer med konstruktører ved nedrivning
- Du vil undgå at skulle lave konstruktører for alle mulige anvendelses scenarier
- Fordi Setter property er mere sigende i XML filerne

Konstruktør og setter property injection kan kombineres



Hvor meget skal man wire?

Hvad skal man wire/konfigurere igennem en BeanFactory?

- Controller delen i MVC (undtagelse for Spring MVC)
- Facader/Services
- DAO'er
- Resourcer (JMS/DataSource/Hibernate Session Factory/JCA/osv.)
- Infrastruktur objekter (sikkerheds controllere, transaktionsmanagere, osv.)
- Dekorative objekt services (transaktions proxies, etc.)
- Definitioner du generelt ønsker eksternaliseret

Det er en klar fejl at konfigurere alle objekter i din applikation igennem en BeanFactory.
Vi skal ikke til at "kode" i XML.



Spring Context

Spring Context tilbyder en specialisering af BeanFactory, kaldet **ApplicationContext**, der tilbyder:

- Internationalisering (MessageSource)
- Event publishing (ApplicationListener)
- Adgang til resources (Filer/URL'er)
- Kan loade flere (hierarkiske) contexts, der kan håndtere hver deres lag
- Auto registrering af BeanPostProcessors og BeanFactoryPostProcessors

ApplicationContext bør bruges fremfor BeanFactory

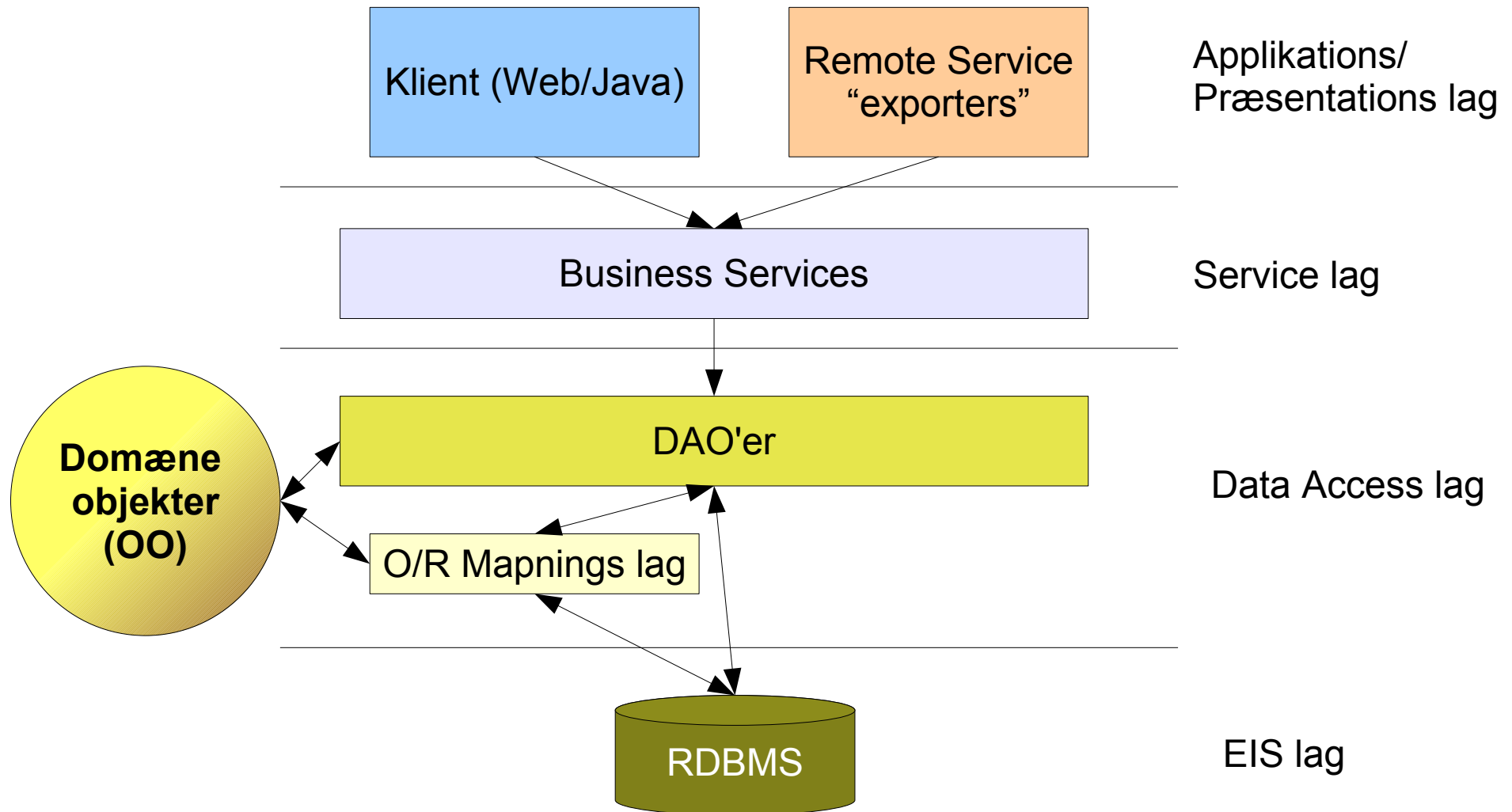


Spring Demo Applikation

Vil vil gennemgå en Spring Demo Applikation der viser hvordan man nemt kan lave en J2EE applikation uden brug af EJB'ere.

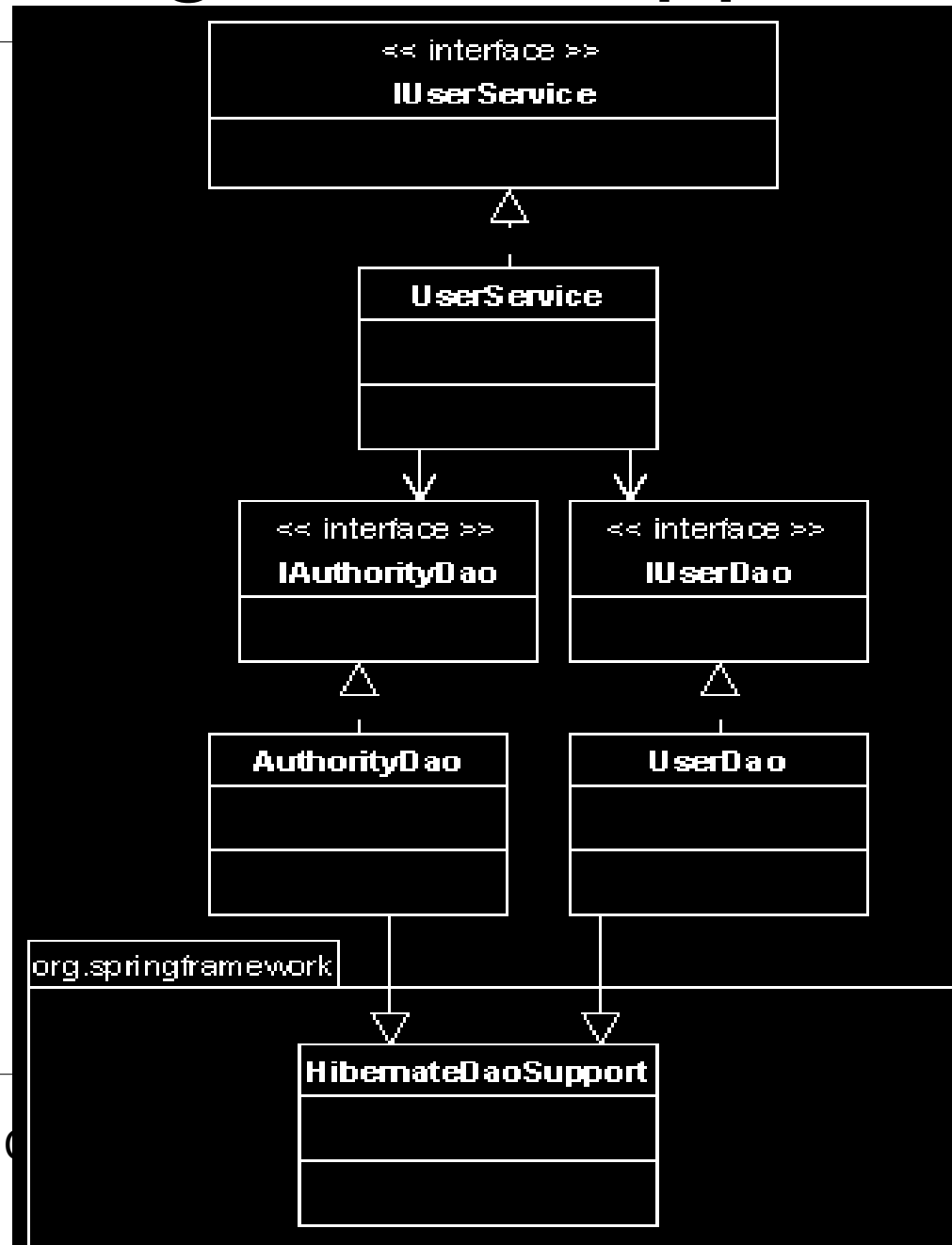


Spring Demo Applikation





Spring Demo Applikation





Spring Service abstraktioner

- **Service abstraktioner:**
 - **Transaktioner**, gennem `TransactionManager` konceptet kan du vælge mellem **globale** transaktioner (JTA) eller **lokale** transaktioner (JDBC, Hibernate,..).
 - **Data Access** (`JdbcTemplate/HibernateTemplate`, etc)
 - **Data Access Exception hierarki**
 - **Mail**
 - **Messaging**



Spring Data Access

- Data Access Object (DAO) pattern'et (Core J2EE Patterns) bruges til at sikre adskillelse mellem forretnings logik og persistens kode
- Spring tilbyder forskellige **DAO Templates** der implementerer den kedelige og repetitive kode som normalt følger med (specielt) JDBC, Hibernate, JDO, osv.
 - Fordele ved Springs DAO Templates
 - DAO Templates sørger for at åbne/deltage i eksisterende Connections, Sessions, etc. samt efter behov at lukke dem bagefter
 - Arbejder med IoC princippet gennem **Callbacks**
 - Sørger for oversættelse af produkt specifikke exceptions til Spring DataAccessExceptions



Spring DAO Templates

- Eksempel på anvendelse af HibernateTemplate

HibernateTemplate

```
public Collection<User> findAll() {
    return (Collection<User>) getHibernateTemplate().execute(
        new HibernateCallback() {
            public Object doInHibernate(Session session) throws HibernateException, SQLException {
                return session.createQuery("from " + User.class.getName()).list();
            }
        }
    );
}
```

Callback

IoC in Action ;-)

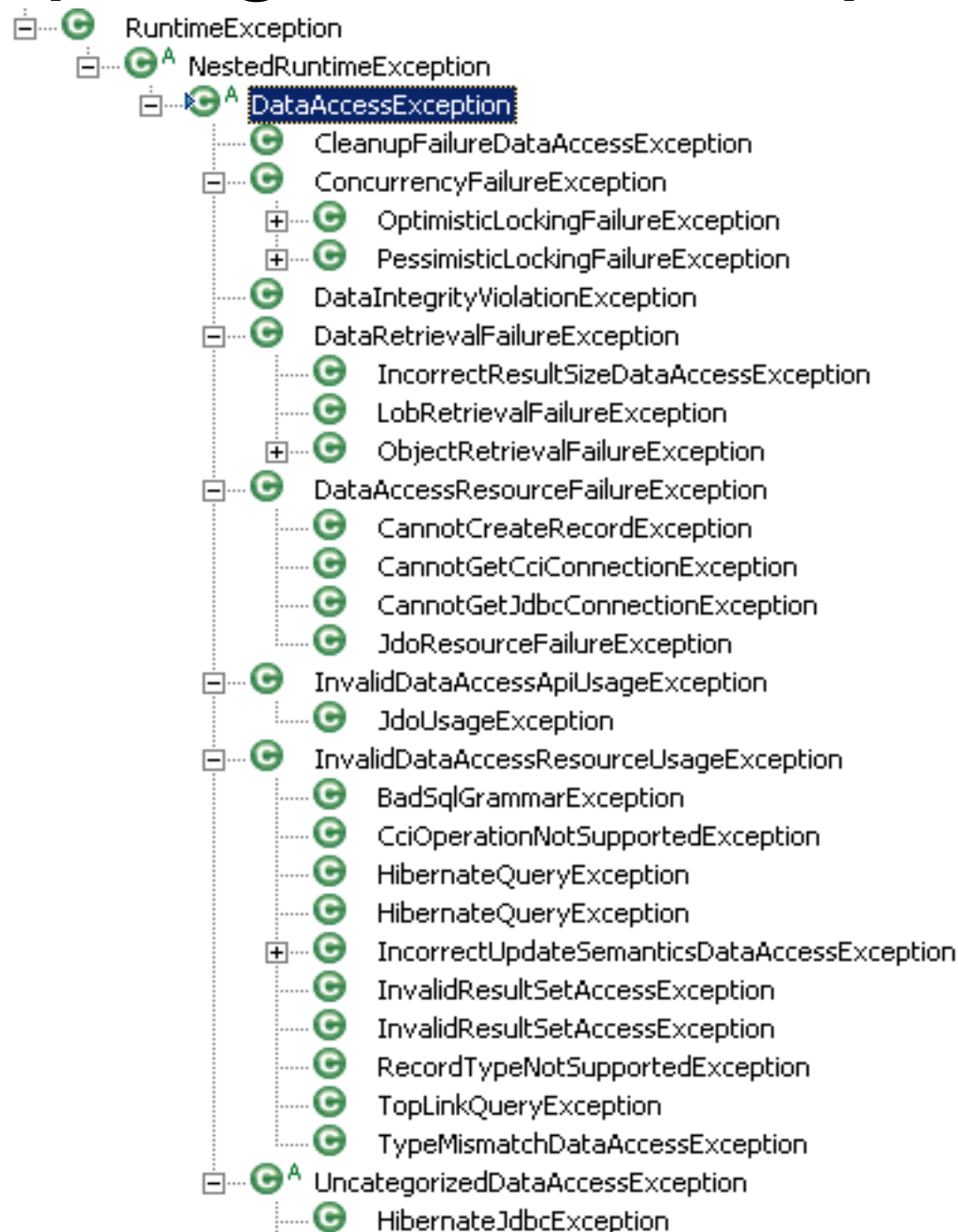


Spring Data Access Exception

- Spring indeholder et komplet `DataAccessException` hierarki der sikrer uafhængighed af persistens implementation.
- Spring oversætter fra DB specifikke fejlkoder til `DataAccessException` (kan nemt udvides med nye databaser)
- `DataAccessException` er **runtime** exceptions, da man under normale omstændigheder ikke kan gøre noget ved dem



Spring DAO Exception





Spring Demo Applikation

Konfiguration af HibernateTransactionManager

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
  ....
  <property name="configLocation">
    <value>hibernate.cfg.xml</value>
  </property>
  <property name="configurationClass">
    <value>org.hibernate.cfg.AnnotationConfiguration</value>
  </property>
</bean>
```



Spring Demo Applikation

Konfiguration af DataSource og tilhørende property configurerer

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
</bean>

<bean id="propertyConfigurer"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
    <list>
      <value>jdbc.properties</value>
    </list>
  </property>
</bean>
```

jdbc.properties

```
jdbc.driverClassName=org.postgresql.Driver
jdbc.url=jdbc:postgresql://localhost/springdemo
jdbc.username=root
jdbc.password=root
```



Spring Demo Applikation

- Springs **deklarative services** (f.eks. transaktioner) benytter Springs Aspect Oriented Programming (AOP) faciliteter
- Det er ikke nødvendigt at forstå AOP for at kunne bruge Springs forskellige deklarative services, men det hjælper med til at forstå hvordan det hele fungerer og hænger sammen



Spring Demo Applikation – Hvad er AOP?

Kode der er spredt, men rent logisk ikke burde være det, går under fælles betegnelse “**Cross Cutting Concerns**”

Aspect Oriented Programming (AOP) tillader dig at samle Cross Cutting Concerns, på i Aspecter.

Derefter kan du deklarativt definere hvor Aspectet skal bruges, uden at ændre en linie i de klasser hvor Aspectet bruges.



Spring Demo Applikation – Hvad er AOP?

Eksempler på Aspecter:

- Transaktioner
- Sikkerhed
- Logging
- Caching
- Auditing
- Exception håndtering
- Retry



Spring AOP – AOP koncepter

AOP koncepter:

- **Aspect** – Aspect er andet ord for Concern. Et Aspect er en modulisering af et koncept, så som transaktion
- **Advice** – Implementationen af et Aspect (*hvad*)
- **JoinPoint** – Et punkt under eksekveringen af et program. Dette vil oftest være en et metode kald (Spring understøtter kun Method JoinPoints)
- **PointCut** – Et sæt af JointPoints, der definerer hvornår et Advice skal udføres (*hvor*)
- **Advisor** – En kombination af et Advice samt PointCuts der definerer hvor Advicet skal udføres (Spring terminologi)



Spring AOP – AOP koncepter

AOP koncepter fortsat:

- **Introduction** – En udvidelse af et objekt med f.eks. nye felter eller metoder
- **Target** – Den klasse der indeholder JointPoints'ene. Også kendt som “advised objekt” eller “proxied objekt”.
- **Proxy** – Resultat objektet, som er en kombination af target objektet og de advices der er bundet til objektet gennem PointCuts.
- **Weaving** – Processen hvor Advices og target objekter bliver kombineret til advised objekter. Kan ske compile time (AspectJ/AspectWerkz) eller Runtime (Spring/AspectWerkz/JBossAOP/etc.)



Spring Demo Applikation

UserService gøres deklarativ transaktionel vha. JDK 5 Annotations og Spring AOP.

```
public interface IUserService {
    /**
     * Find all users
     * @return Collection of users
     */
    @Transactional(propagation=Propagation.REQUIRED, readOnly=true)
    Collection<User> findAllUsers();

    /**
     * Create a new user
     * @param user The context containing info about the new user to create
     * @return The User object
     * @throws NoSuchAuthorityException In case on of the supplied authorities
     *         doesn't exist
     */
    @Transactional(propagation=Propagation.REQUIRED,
        rollbackFor=NoSuchAuthorityException.class)
    User createUser(CreateUserContext user) throws NoSuchAuthorityException;
}
```



Spring Demo Applikation

Konfiguration af deklarative services

```
<!-- This bean is a bean post-processor that will automatically apply relevant advisors
      to any bean in child factories -->
<bean
    class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<!-- AOP advisor that will provide declarative transaction management based on
      attributes. -->
<bean
class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
    <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<!-- Transaction interceptor to use for auto-proxy creation (JDK 5 annotations) -->
<bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributeSource">
        <bean
class="org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"/>
        </property>
    </bean>
```



Spring Demo Applikation

Alternativ måde at gøre UserService transaktionel

```
<bean id="userService"  
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">  
  <property name="transactionManager" ref="transactionManager"/>  
  <property name="target">  
    <bean class="dk.cramon.springdemo.service.UserService"  
      autowire="constructor" />  
  </property>  
  <property name="transactionAttributes">  
    <props>  
      <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>  
      <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>  
      <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>  
      <prop key="*">PROPAGATION_REQUIRED</prop>  
    </props>  
  </property>  
</bean>
```



Spring Demo Applikation

- Message Driven POJO (MDP) med ActiveMQ
 - Kort gennemgang af hvordan man laver MDP'er i Spring baseret på ActiveMQ
 - MDP understøttelse kommer som en del af Spring 1.3, men indtil videre er dokumentationen sparsom, derfor bruger vi ActiveMQ's tilgang til MDP'er istedet.
 - Hvis tiden tillader det en kort gennemgang af hvordan man laver man laver asynkrone beans (ala Indigo) med ActiveMQ og Lingo.



Message Driven POJO (MDP)

Vores `AuditListener` MDP skal implementere `MessageListener` interfacet

```
public class AuditListener implements MessageListener {  
  
    public AuditListener() {  
        super();  
    }  
  
    public void onMessage(Message message) {  
        ...  
    }  
}
```



MDP Konfiguration af ActiveMQ

Vi definerer en JCA Container så vores MDP kan kommunikere med JMS serveren

```
<bean id="jcaContainer" class="org.activemq.jca.JCAContainer">
  <property name="workManager">
    <bean class="org.activemq.work.SpringWorkManager" />
  </property>
  <property name="resourceAdapter">
    <bean class="org.activemq.ra.ActiveMQResourceAdapter">
      <property name="serverUrl"
        value="tcp://localhost:61616" />
    </bean>
  </property>
</bean>
```



MDP konfiguration

Vi definerer at vores AuditListener objekt skal connecte til SpringDemo.AuditQueue køen igennem JCA Containeren.

```
<bean id="auditListener"
      class="dk.cramon.springdemo.mdp.AuditListener" />

<bean id="auditMdp" factory-method="addConnector"
      factory-bean="jcaContainer">
  <property name="activationSpec">
    <bean class="org.activemq.ra.ActiveMQActivationSpec">
      <property name="destination"
                value="SpringDemo.AuditQueue" />
      <property name="destinationType"
                value="javax.jms.Queue" />
    </bean>
  </property>
  <property name="ref" value="auditListener" />
</bean>
```



MDP kommunikation via JMS

Kommunikation til MDP'en foregår via Springs JMSTemplate

```
public class AuditAdvice implements MethodInterceptor {
    private JmsTemplate auditJmsTemplate;
    public AuditAdvice(JmsTemplate auditJmsTemplate) {
        this.auditJmsTemplate = auditJmsTemplate;
    }

    ...

    protected void auditMessage(final String serviceName, final String
                                operationName, final boolean success) {
        auditJmsTemplate.send(new MessageCreator() {
            public Message createMessage(Session session)
            throws JMSEException {
                MapMessage message = session.createMapMessage();
                message.setString("serviceName", serviceName);
                message.setString("operationName", operationName);
                message.setBoolean("success", success);
                message.setLong("timeStamp", System.currentTimeMillis());
                return message;
            }
        });
    }
}
```



MDP – Konfiguration af JMSTemplate

JMSTemplaten sender via et ConnectionFactory messages til SpringDemo.AuditQueue'en

```
<bean id="auditJmsTemplate"
  class="org.springframework.jms.core.JmsTemplate">
  <property name="defaultDestinationName"
    value="SpringDemo.AuditQueue" />
  <property name="connectionFactory" ref="connectionFactory" />
</bean>

<bean id="connectionFactory"
  class="org.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
```



Spring Schema Extension

- Springs generelle BeanFactory/ApplicationContext XML konfigurations sprog er et lavniveau sprog.
- Et lavniveau sprog giver den fleksibilitet som er nødvendigt for at kunne håndtere alle konfigurations opgaver.
- Dette sker på bekostning af udtryks graden og dermed størrelsen af konfigurations filerne.
- Et lavniveau sprog er ikke så udtryksfuldt som et højniveau sprog.



Spring Schema Extension

- Et højniveau sprog kan udtrykke megen information på lidt plads modsat et lav niveau sprog.
- Tilgængæld kan et højniveau sprog ikke håndtere lige så mange forskellige typer opgaver som et lav niveau sprog.
- Med højniveau sprog giver man afkøb på fleksibilitet til gengæld for at effektivitet.
- Højniveau sprog kan også kaldes for et Domæne Specifikke Sprog (DSL)
- Spring Schema Extension er mit bud på at hvordan man kan definere et høj niveau sprog til konfiguration af Spring's BeanFactory/ApplicationContext, som kan genbruge Springs generelle konfigurations sprog.

Spring MVC med standard sprog

```
<bean id="viewResolver"  
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>  
    <property name="prefix" value="/WEB-INF/jsp/" />  
    <property name="suffix" value=".jsp"/>  
</bean>  
<bean id="defaultHandlerMapping"  
    class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />  
  
<bean name="/shop/addItemToCart.do"  
    class="org.springframework.samples.jspetstore.web.spring.AddItemToCartController">  
    ....  
</bean>
```

```
<bean id="urlMapping"  
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">  
    <property name="mappings">  
        <props>  
            <prop key="/shop/addToItemCart.do">addItemToCartController </prop>  
            ....  
        </props>  
    </property>  
</bean>  
<bean id="addItemToCartController"  
    class="org.springframework.samples.jspetstore.web.spring.AddItemToCartController">  
    <property name="successView" value="Cart" />  
    <property name="formView" value="Cart" />  
    <property name="petStore" ref="petStore" />  
</bean>
```



Spring MVC – Schema Extension

```
<spring-mvc xmlns="http://www.myproject.com/springmvc"
            xmlns:beans="http://www.springframework.org/beans" ....>

  <view-definition
    viewClass="org.springframework.web.servlet.view.JstlView"
    prefix="/WEB-INF/jsp" suffix=".jsp" />

  <action path="/shop/addItemToCart.do"
    class="org.springframework...AddItemToCartController">
    <forward name="formView">Cart</forward>
    <forward name="successView">Cart</forward>
    <beans:property name="petStore" ref="petStore" />
  </action>

  <action path="/shop/newAccount.do"
    class="org.springframework...AccountFormController">
    <forward name="success">index</forward>
    <beans:property name="petStore" ref="petStore" />
    <beans:property name="validator" ref="accountValidator" />
  </action>
</spring-mvc>
```



Standard Spring – Definitions genbrug

```
<bean id="baseTransactionProxy"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
      abstract="true">
  <property name="transactionManager">
    <ref bean="transactionManager" />
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>

<bean id="petStore" parent="baseTransactionProxy">
  <property name="target">
    <bean class="org.springframework.samples.jpetsy.domain.logic.PetStoreImpl">
      <property name="accountDao" ref="accountDao" />
      <property name="categoryDao" ref="categoryDao" />
      <property name="productDao" ref="productDao" />
      <property name="itemDao" ref="itemDao" />
      <property name="orderDao" ref="orderDao" />
    </bean>
  </property>
</bean>
```



Spring Schema Extension

```
<beans xmlns="http://www.springframework.org/beans"
  xmlns:transaction="http://www.springframework.org/transaction"
  xmlns:springData="http://www.springframework.org/data" ...>

  <transaction:transactional defaultPropagation="PROPAGATION_REQUIRED"
    transactionManager="transactionManager">
    <bean class="org.springframework....logic.PetStoreImpl">
      <property name="accountDao" ref="accountDao" />
      <property name="categoryDao" ref="categoryDao" />
      <property name="productDao" ref="productDao" />
      <property name="itemDao" ref="itemDao" />
      <property name="orderDao" ref="orderDao" />
    </bean>
    <bean class="org.springframework.....logic.OtherFacade">
      ...
    </bean>
  </transaction:transactional>

  <springData:dao id="accountDao" class="....AccountDao">
    <constructor-arg>
      <ref bean="dataSource" />
    </constructor-arg>
  </springData:dao>

</beans>
```



Spring Schema Extension implementation

- Hvert namespace har sin egen `SchemaExtensionParser`, som samarbejder med Springs `BeanDefinitionParser` og `BeanDefinitionReader`.
- `SchemaExtensionParser` er en event drevet parser (ala SAX), der arbejder med navngivne parse metode kald, `start<ElementNavn>` og `end<ElementNavn>`, for hvert element `BeanDefinitionParser`en møder.



Spring Schema Extension implementation

```
public class MyProjectDataAccessParser implements SchemaExtensionParser {

    public MyProjectDataAccessParser() {
        super();
    }

    ...

    public void startDao(Element element, ElementParseContext parseContext) {
        String id = element.getAttribute("id");
        String className = element.getAttribute("class");
        AbstractBeanDefinition beanDefinition =
            parseContext.createRootBeanDefinition(className);

        BeanDefinitionHolder beanDefinitionHolder = new
            BeanDefinitionHolder(beanDefinition, id);

        parseContext.registerBean(beanDefinitionHolder);
    }

    public void endDao(ElementParseContext parseContext) {
    }
}
```



Vil du vide mere om Spring Schema Extension...?

Læs mere på

<http://blog.javagruppen.dk/blog/files/Spring.pdf>

eller følg med i diskussionerne om emnet på

<http://www.springframework.org>



Tak for opmærksomheden

Spørgsmål?

Ellers er I velkomne til at kontakte mig efter foredraget eller via mail på jeppe@cramon.dk,