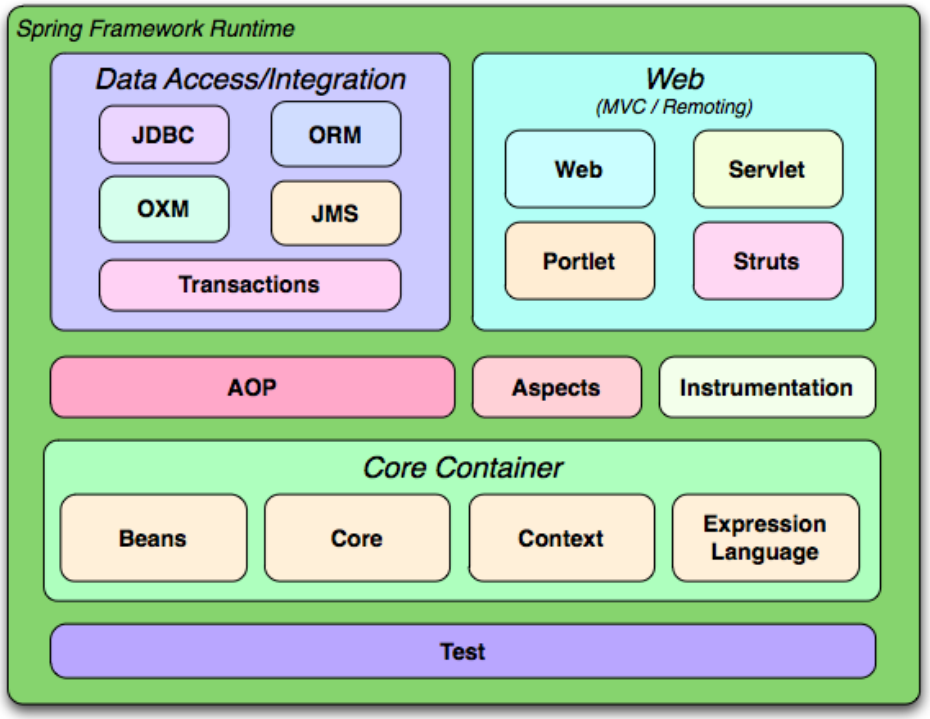
Présentation Spring

Max Devulder

Sommaire

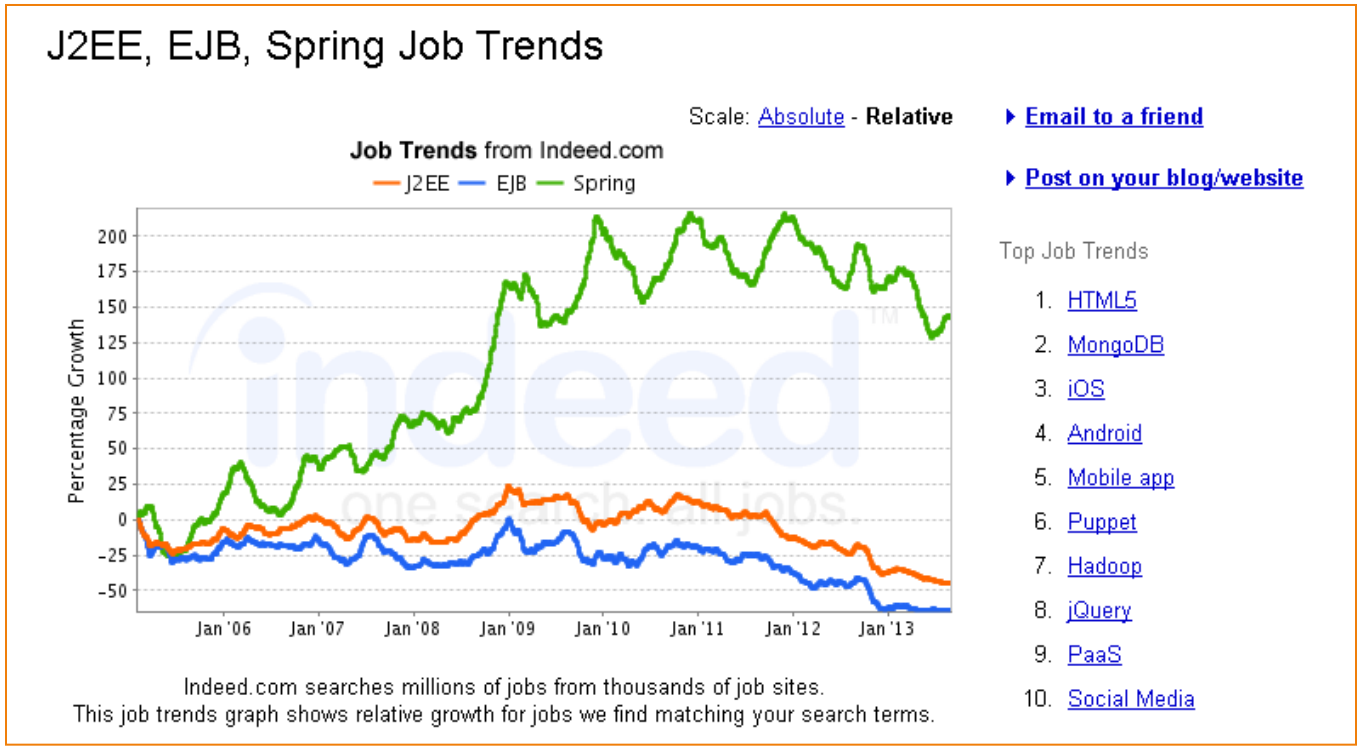
- **Framework** de développement basé sur la notion de « conteneur léger »
- Projet Open Source, support & évolution par la société SpringSource
- Spring est composé de **briques,** isolées & utilisables à souhait

- Liste des modules Spring 3.0 (20 modules)



http://docs.spring.io/spring/docs/3.0.x/reference/overview.html

- Très utilisé pour les nouveaux projets depuis 2009 (indeed.com)



http://www.indeed.com/jobtrends?q=J2EE%2C+EJB%2C+Spring&relative=1&relative=1

Spring Core : Implémente notamment le concept d'inversion de contrôle (injection de dépendance). Il est également responsable de la gestion et de la configuration du conteneur.

Spring Context : Ce module étend Spring Core. Il fournit une sorte de base de données d'objets, permet de charger des ressources (telles que des fichiers de configuration) ou encore la propagation d'évènements et la création de contexte comme par exemple le support de Spring dans un conteneur de Servlet.

Spring AOP : Permet d'intégrer de la programmation orientée aspect.

Spring DAO : Ce module permet d'abstraire les accès à la base de données, d'éliminer le code redondant. Il fournit en outre une gestion des transactions.

Spring MVC : Equivalent de Struts.

Sommaire

- De nombreux problèmes sont récurrents durant la phase de conception.
- Les Design Pattern (patron de conception) apportent les **meilleurs solutions** à ces problèmes.
- Les Pattern de Gof (Gang of four - Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides) sont aux nombre de **23**.
- Spring IOC en utilise principalement deux : le Singleton & la Fabrique

Citations
« Les patrons offrent la possibilité de capitaliser un savoir précieux né du savoir-faire d'experts »

Le Singleton
Un objet = 1 instance !

- Il est inutile d'instancié plusieurs fois un service !

- La solution :

1. Un constructeur private ()
   => Personne ne me voit !
2. Une instance private statique
   => Je suis responsable de mon **unique** instance.
3. Une méthode publique
   => Je décide comment la partager au monde entier.

```
public class MyFirstSingleton {

    private static MyFirstSingleton instance;

    private MyFirstSingleton() {
        // Singleton cannot be instantiate !
    }

    /**
     * Hello world, you want me ?
     * @return
     */
    public MyFirstSingleton getInstance() {
        if (instance == null) {
            instance = new MyFirstSingleton();
        }
        return instance;
    }
}
```

La Fabrique

Vos baskets sont fabriquées en usine, pourquoi pas vos objets ?

- Je veux changer de SGBD régulièrement !

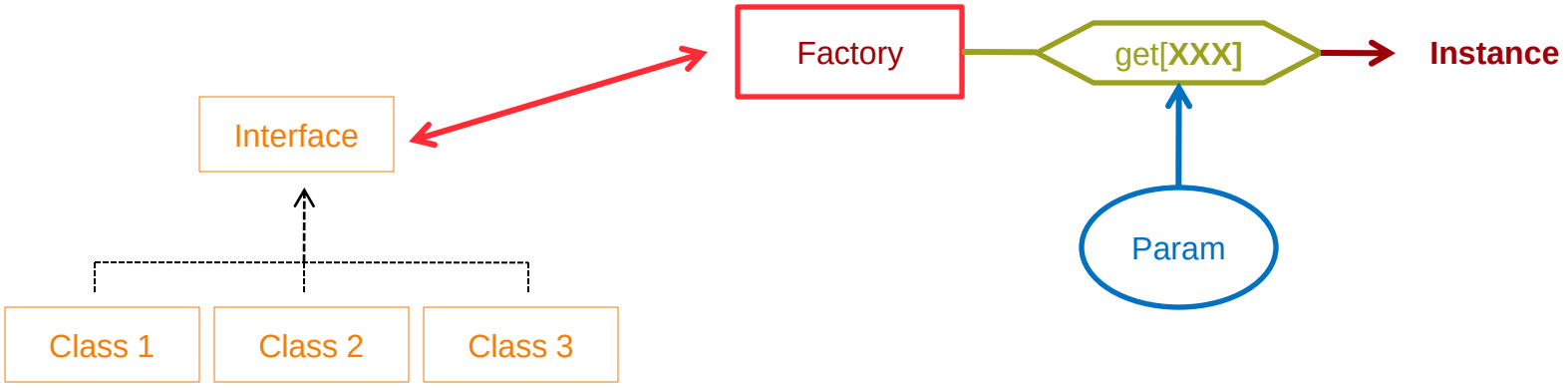
- La solution :

1. Polymorphisme => **1** Interface, **X** Classes

Bonjour, je gère les SGBD, que désirez vous ?

2. Des identifiants = des codes barres.

« MySql » s'il vous plait.



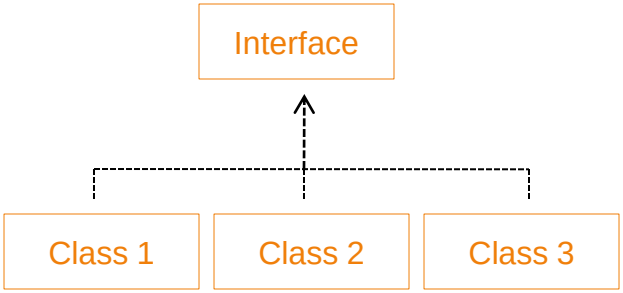
La Fabrique

Vos baskets sont fabriquées en usine, pourquoi pas vos objets ?

- Concrètement

Je regroupe mes beans par une interface

```
import java.sql.Connection;

public interface DataBaseConnection {
    /**
     *
     * @return Opened connection to the BD !
     */
    Connection getConnect() throws Exception;
}
```



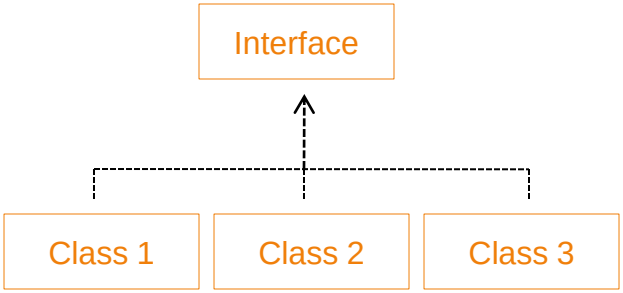
La Fabrique

Vos baskets sont fabriquées en usine, pourquoi pas vos objets ?

Concrètement

Je regroupe mes beans par une interface

```
import java.sql.Connection;
import java.sql.SQLException;

import oracle.jdbc.pool.OracleDataSource;

public class OracleConnection implements DataBaseConnection {

    public Connection getConnect() {

        try {
            // Create a OracleDataSource instance explicitly
            final OracleDataSource ods = new OracleDataSource();
            ods.setUser("scott");
            ods.setPassword("tiger");
            ods.setDriverType("oci8");
            ods.setNetworkProtocol("ipc");
            return ods.getConnection();
        } catch (final SQLException e) {
            // Something better to do here.
            return null;
        }
    }

}
```



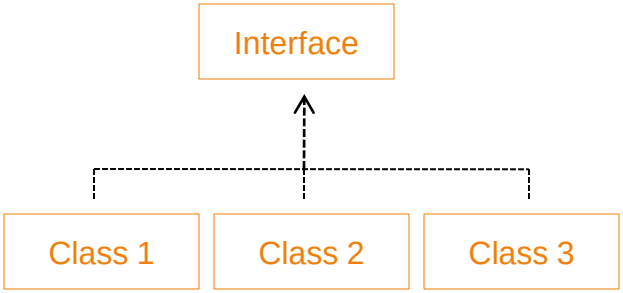
La Fabrique

Vos baskets sont fabriquées en usine, pourquoi pas vos objets ?

Concrètement

Je regroupe mes beans par une interface

```
import java.sql.Connection;

public class MySqlConnection implements DataBaseConnection {

    private Connection conn = null;

    public Connection getConnect() {
        try {
            Class.forName("com.mysql.jdbc.Driver").newInstance();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        try {
            return DriverManager.getConnection("jdbc:mysql:....");
        } catch (SQLException e) {
            // Something better to do here
            e.printStackTrace();
        }
        return conn;
    }

}
```

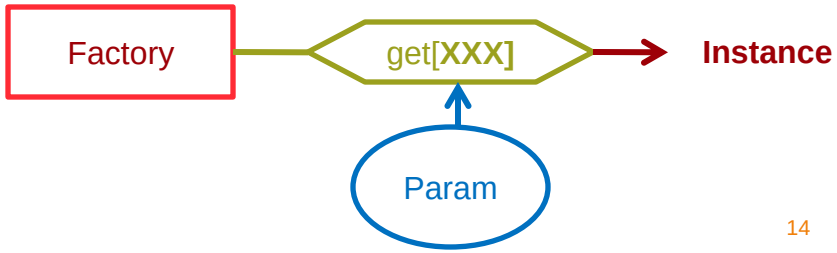


La Fabrique

Vos baskets sont fabriquées en usine, pourquoi pas vos objets ?

Concrètement

```
public class DataBaseFactory {

    public static final String CONNECTION_MYSQL = "MySql";
    public static final String CONNECTION_ORACLE = "Oracle";

    /**
     * Retourne la connection associée à l'id.
     *
     * @param connectionId
     *            : CONNECTION_MYSQL/CONNECTION_ORACLE
     * @return {@link DataBaseConnection} : Implémentation correspondante
     */
    public static DataBaseConnection getDataBaseConnection(final String connectionId) {
        if (CONNECTION_MYSQL.equalsIgnoreCase(connectionId)) {
            return new MySqlConnection();
        } else if (CONNECTION_ORACLE.equalsIgnoreCase(connectionId)) {
            return new OracleConnection();
        } else {
            throw new IllegalStateException("Connection inconnue !");
        }
    }
}
```



La Fabrique

Vos baskets sont fabriquées en usine, pourquoi pas vos objets ?

Avantages

```
public static void main (String args[]) throws Exception{
    // J'ai besoin d'une connection ? => DatabaseFactory

    final DataBaseConnection dbBaseConnection = DataBaseFactory.getDataBaseConnection(DataBaseFactory.CONNECTION_ORACLE);

    // J'utilise oracle ici, si j'ai besoin de MySql, 1 paramètre à changer !
    final Connection connect = dbBaseConnection.getConnect();

}
```

Maintenance	Je peux changer rapidement / facilement de SGBD.
Evolution	L'instanciation de mes objets (new) est centralisée. On veut logger chaque instanciation ? => Facile.
Souplesse	Je peux rajouter un nouveau type de SGBD très facilement.

Sommaire

Les principaux modules
Spring IOC

- IOC « Inversion Of Control » : Facilite l'intégration des composant entre eux.

- Délégation des dépendances entre objets dans via fichier xml :applicationContext.xml
  (Rappel :Conteneur léger = non intrusif)

Exemple simple d'utilisation

SpringContext.xml

```
<beans>
  <bean id="produitDao" class="com.monsiteecommerce.dao.ProduitDao" scope="singleton">
  </bean>
  <bean id="catalogueProduitBo" class ="com.monsiteecommerce.bo.CatalogueBo" scope="singleton">
    <property name="myDaoProduit"> <ref bean="produitDao"/> </property>
  </bean>
</beans>
```

CatalogueProduitBo.java

```
public class CatalogueProduitBo implement ICatalogueProduitBo{

   private IProduitDao produitDao;

   public setProduitDao (final IProduitDao produitDao){
      this.produitDao = produitDao;
   }
}
```

Automatiquement instancié par Spring au démarrage du serveur.

Il existe 2 types d'injection

- Par mutateur (setter)
- Par constructeur

Mutateur

ExampleBean.java

```java
package examples;

public class ExampleBean {

  private AnotherBean beanOne;

  private YetAnotherBean beanTwo;

  public void setBeanOne(AnotherBean beanOne) {
    this.beanOne = beanOne;
  }

  public void setBeanTwo(YetAnotherBean beanTwo) {
    this.beanTwo = beanTwo;
  }
}
```

ApplicationContext.java

```xml
<bean id="exampleBean" class="examples.ExampleBean">

   <!-- injection par setter en utilisant la balise <ref/> -->
  <property name="beanOne">
    <ref bean="anotherExampleBean" />
  </property>

  <!-- injection par setter en utilisant l'attribut 'ref' -->
  <property name="beanTwo" ref="yetAnotherBean" />
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean" />
<bean id="yetAnotherBean" class="examples.YetAnotherBean" />
```

- Il existe 2 types d'injection
  - Par mutateur (setter)
  - Par constructeur

- Constructeur

ExampleBean.java

```
package examples;

public class ExampleBean {

  private AnotherBean beanOne;

  private YetAnotherBean beanTwo;

   public ExampleBean(AnotherBean anotherBean, YetAnotherBean
   yetAnotherBean) {
     this.beanOne = anotherBean;
     this.beanTwo = yetAnotherBean;
   }
}
```

ApplicationContext.java

```
<bean id="exampleBean" class="examples.ExampleBean">
   <constructor-arg index="0"><ref bean=" anotherExampleBean "/></constructor-arg>
   <constructor-arg index="1"><ref bean=" yetAnotherBean "/></constructor-arg>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean" />
<bean id="yetAnotherBean" class="examples.YetAnotherBean" />
```

Principales propriétés : bean

Attribut/Elément	Description
Class	Chemin complet vers la classe à instancier. Obligatoire
Name	Identifiant qui sera utilisé pour désigner cette classe Obligatoire Doit être unique.
Scope	Scope de l'objet « Singleton » : Une seule instance (par défaut) « Prototype » : Multi instance.
Constructor-arg	Injection via constructeur (Cf. slide précédent)
Property	Injection via mutateur (Cf. slide précédent)
Lazy-init	Instanciation à quel moment ? « True » : au premier accès à la classe. « False » : au démarrage du server
Init-method	Appel d'une méthode juste après l'instanciation de l'objet
Destroy-method	Appel d'une méthode juste avant la destruction de l'objet.

Par annotation (depuis Spring 2.5)

Annotation	Description
@Resource	Déclaration d'une ressource qui nécessite injection
@Autowired	Injecte la dépendance « by-type »
@Qualifier	Permet à Spring de choisir la bonne dépendance
@Required	Dépendance obligatoire

Exemple

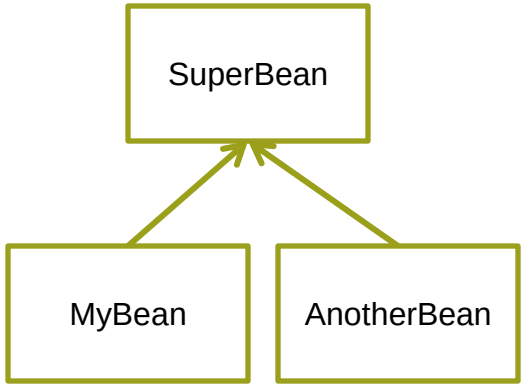
ExampleBean.java

```
package examples;

@Resource
public class ExampleBean {

  @Autowired
  private AnotherBean beanOne;
}
```

A-t-on encore besoin de fichier XML ?

ExampleBean.java

```
package examples;

@Resource
public class ExampleBean {

  @Autowired
  @Qualifier("bean1")
  private AnotherBean beanOne;

}
```



ApplicationContext.java

```
<bean id="superBean" class="examples.SuperBean" />
<bean id="bean1" class="examples.AnotherBean"  parent= "superBean"/>
<bean id="bean2" class="examples.MyBean"  parent= "superBean"/>
```