

Spring MVC

Mohamed DERKAOU

13/03/2018

Plus d'informations sur <http://www.dawan.fr>
Contactez notre service commercial au **0800.10.10.97** (appel gratuit depuis un poste fixe)

Plan



- Introduction : Java EE, Spring
- Configurer des beans avec Spring Core
- Implémenter des contrôleurs avec Spring MVC
- Utiliser les espaces de persistance

Java EE



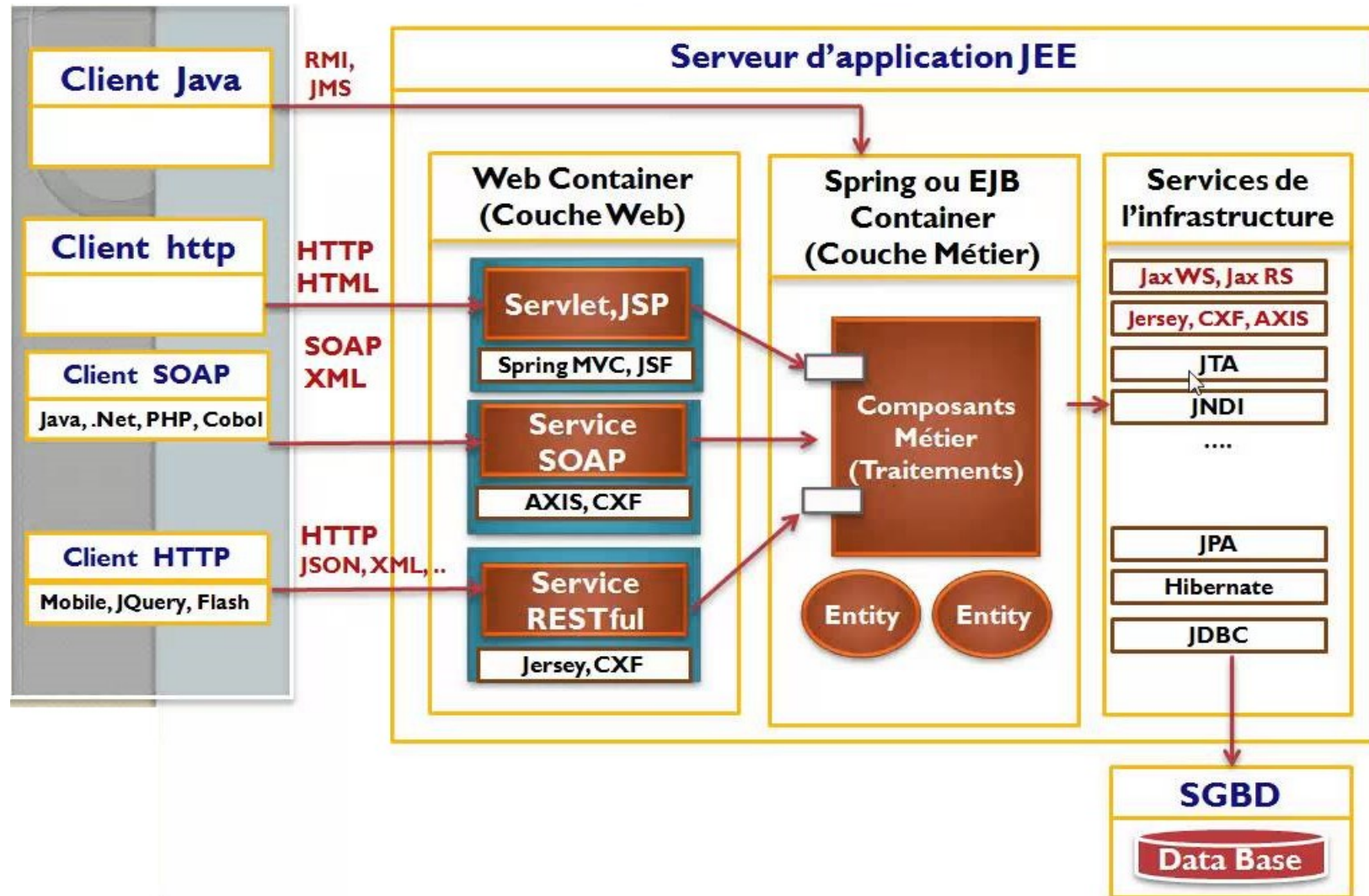
- Java EE est la version "entreprise" de Java, elle a pour but de faciliter le développement d'applications distribuées.
Mais en fait, Java EE est avant tout une norme.
- C'est un ensemble de standard décrivant des services techniques comme, par exemple, comment accéder à un annuaire, à une base de données, à des documents...

Important : Java EE définit ce qui doit être fournit mais ne dit pas comment cela doit être fournit.

Exemple de services :

- JNDI (Java Naming and Directory Interface) est une API d'accès aux services de nommage et aux annuaires d'entreprises tels que DNS, NIS, LDAP...
- JTA (Java Transaction API) est une API définissant des interfaces standard avec un gestionnaire de transactions.

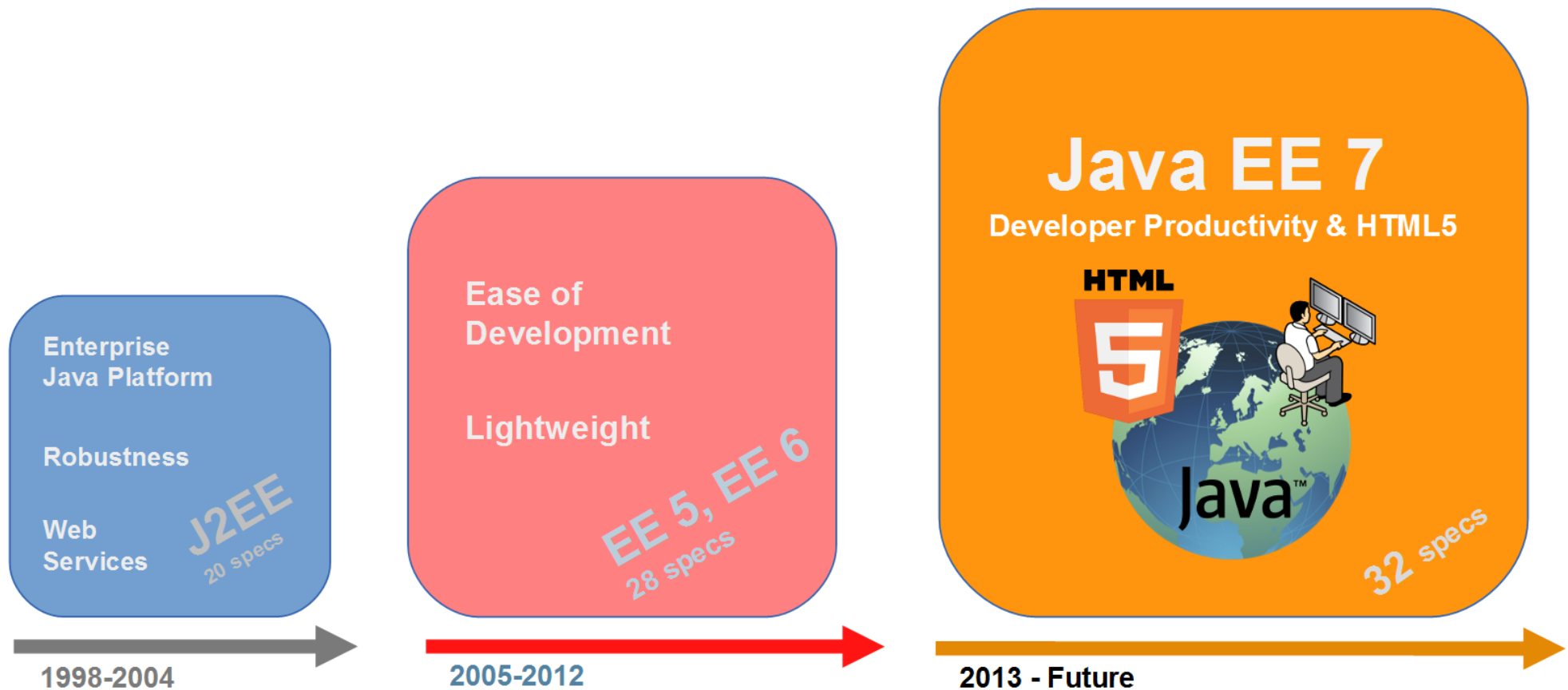
Architecture JEE



Serveurs d'applications JEE

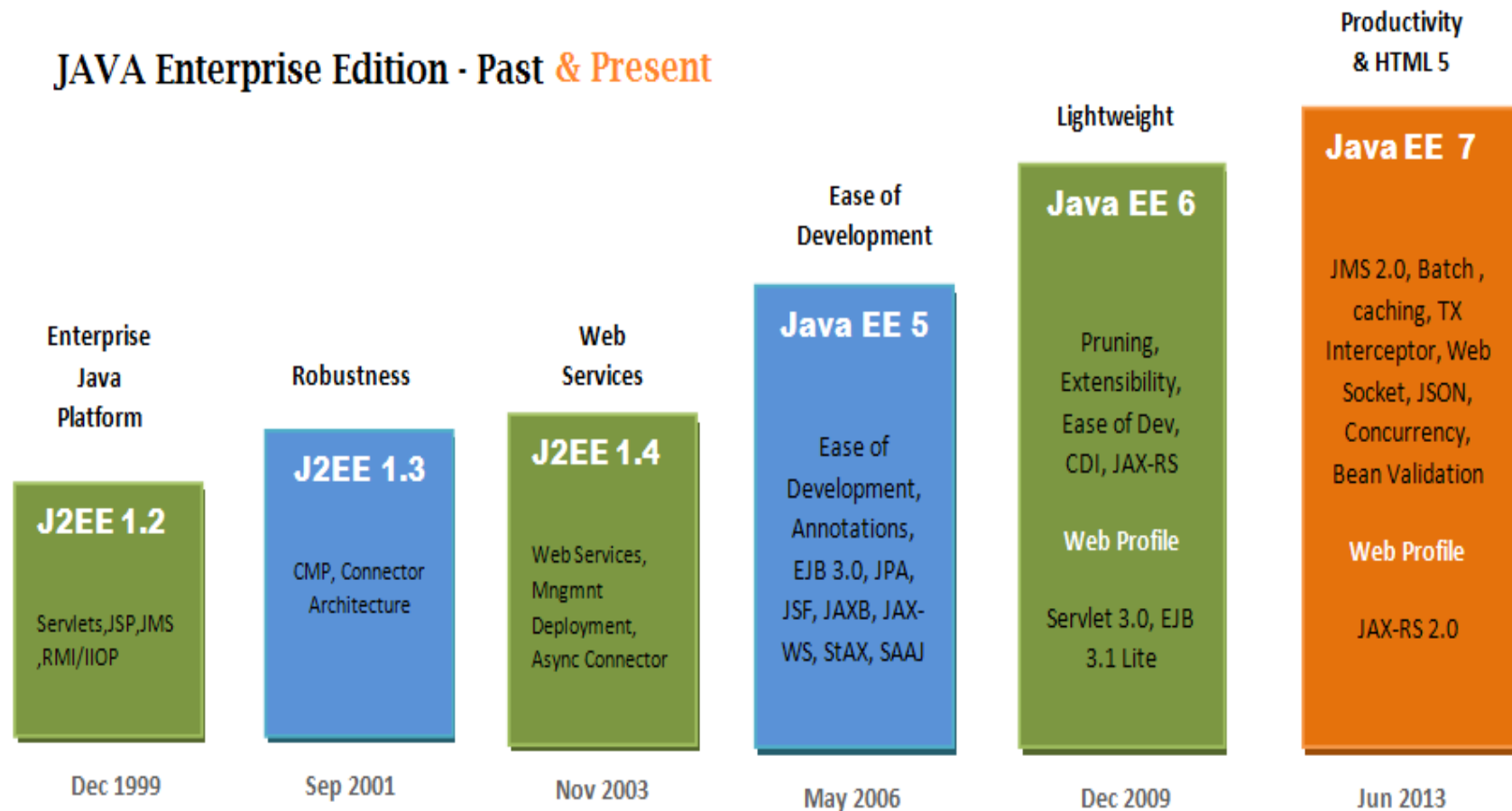
- Les applications JEE sont hébergées par des serveurs certifiés JEE : Web-Profile ou Full-Profile
<http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html>
- Exemples de serveurs d'applications JEE :
 - Apache Tomcat (conteneur web uniquement)
 - Oracle GlassFish (implémentation de référence) :
[**https://glassfish.java.net/download.html**](https://glassfish.java.net/download.html)
 - Oracle WebLogic
 - IBM WebSphere
 - JBoss
 - ...

Evolution



Evolution (2)

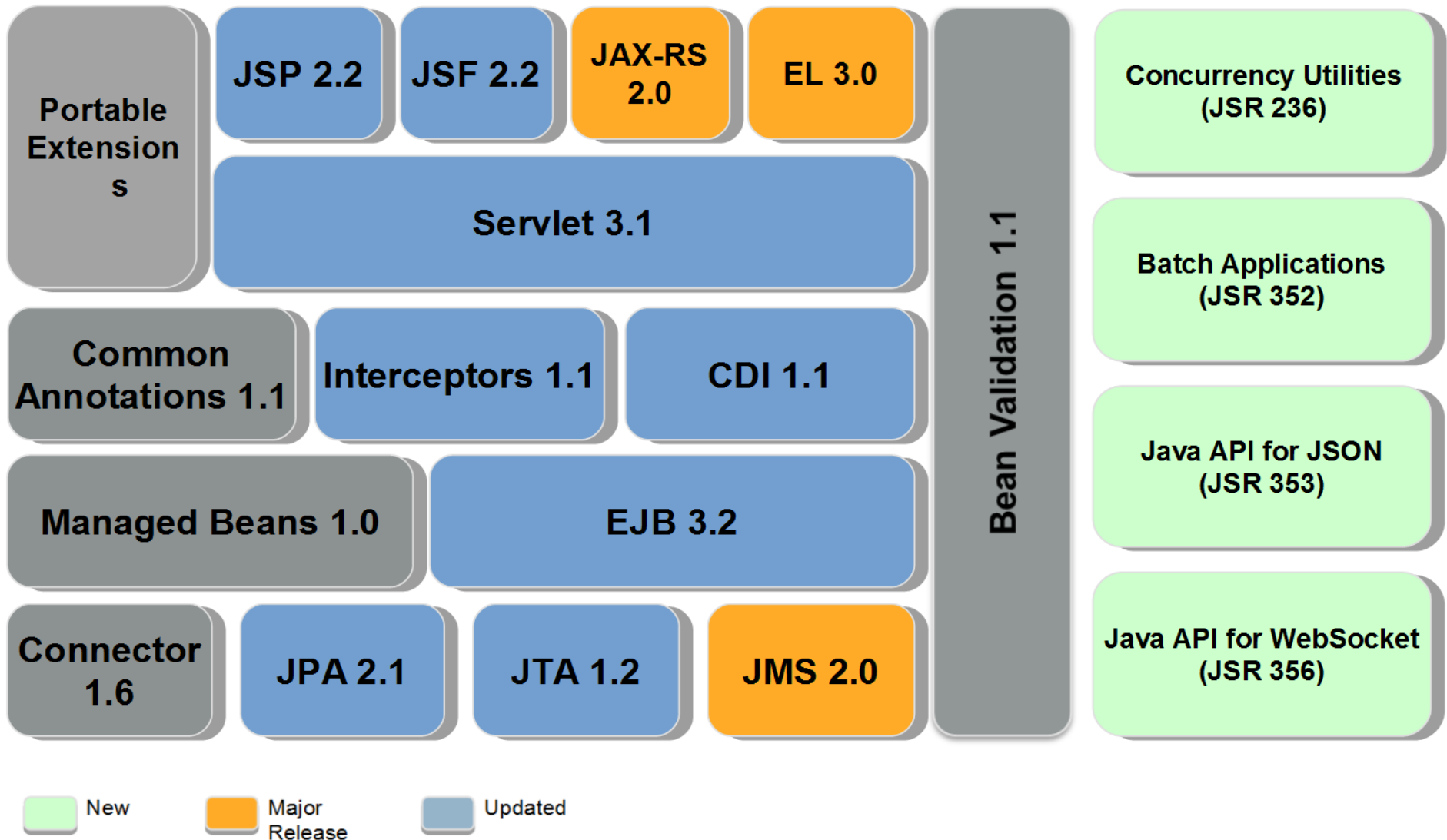
JAVA Enterprise Edition - Past & Present



JEE 7



JEE 7 : APIs



Spring



Concept IoC



- Patron d'architecture qui fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application mais du framework ou de la couche logicielle sous-jacente
- Utilisation la plus connue : l'inversion des dépendances (dependency inversion principle, 1994, Robert C. Martin)
- En POO, loc permet de découpler les dépendances entre objets

Concept IoC



- Avec l'IoC, le framework prend en charge l'exécution principale du programme, il coordonne et contrôle l'activité de l'application
- Le programme utilisateur définit alors les blocs de codes en utilisant l'API fournie à cet effet par le framework, sans relation dure entre eux. Ces blocs de codes sont laissés à la discrétion du framework qui se chargera de les appeler.

Container léger



« SPRING est effectivement un conteneur dit “ léger ”, c’est-à-dire une infrastructure similaire à un serveur d’applications Java EE. Il prend donc en charge la création d’objets et la mise en relation d’objets par l’intermédiaire d’un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets. Le gros avantage par rapport aux serveurs d’application est qu’avec SPRING, les classes n’ont pas besoin d’implémenter une quelconque interface pour être prises en charge par le framework (au contraire des serveur d’applications Java EE et des EJBs). C’est en ce sens que SPRING est qualifié de conteneur “ léger ”. »

Erik Gollot, *Introduction au framework Spring*

Framework IoC



- Spring
- Castle
- PocoCapsule
- Indigo pour Flash, Flex et AIR
- FLOW3, écrit en PHP, qui motorise la version 5 du CMS TYPO3
- Google Guice, écrit en Java
- Drupal CMS
- Design pattern provider du Framework .NET.

Spring



Framework applicatif pour faciliter et améliorer la productivité de développement d'applications.

Il intègre :

- IoC : via l'injection de dépendances
- La programmation orientée aspect
- Une couche d'abstraction

La couche d'abstraction permet d'intégrer facilement des bibliothèques ou des frameworks déjà existants.

Spring



Spring 1 (2004 ?)

Spring 2 (2006) : request&session scope, meilleur xml ou annotations, support de AspectJ, JMS asynchrone, JSP tags, dynamic languages,

Spring 2.5 (2007) : JSF2, Tiles2, Websphere, java 1.4.2, (pas java 6 ? bug classFor())

Spring 3 : 2009

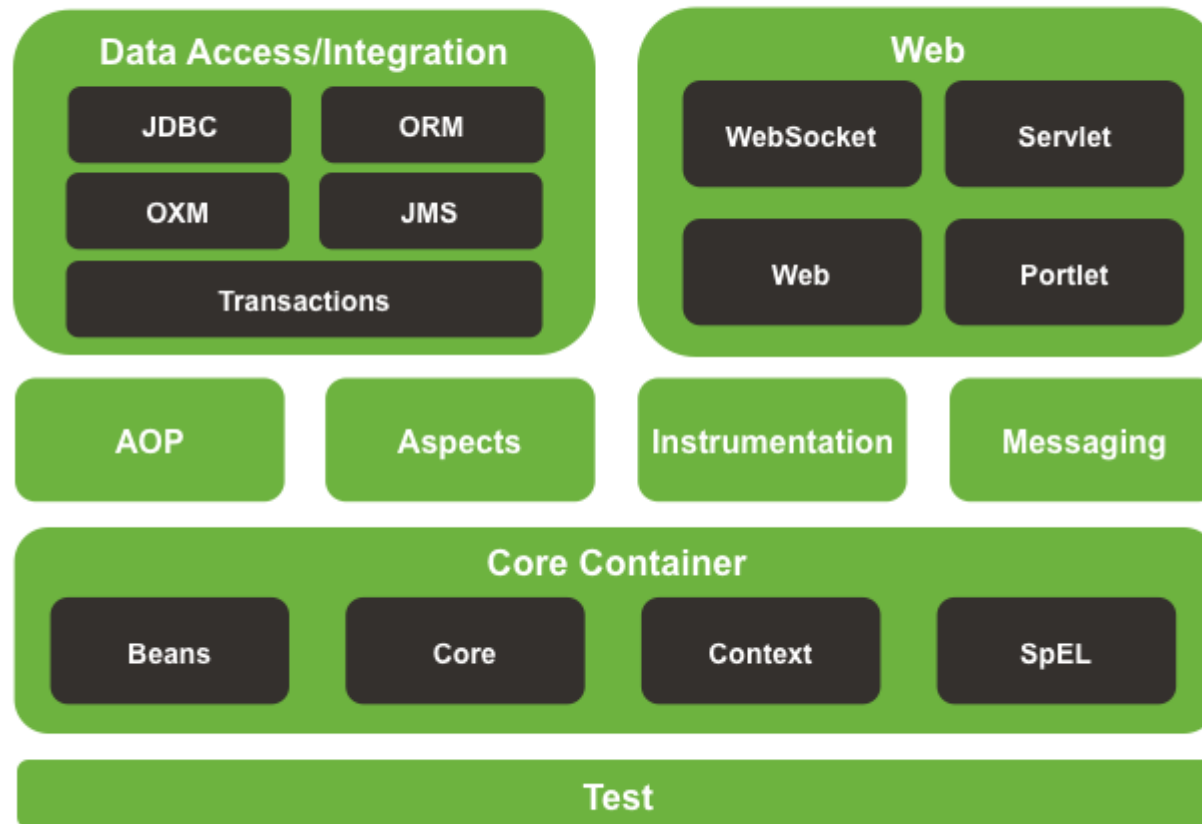
Spring 3.1 : 2011

Spring 4 : 2013 (JavaSE 8, WebSocket, ...)

Structure du framework



Spring Framework Runtime



Galaxie Spring

modules d'architecture applicative



Le cadre de Spring consistait à apporter un conteneur léger servant à l'loc

Aujourd'hui, Spring représente un grand nombre de modules logiciels :

- **Spring framework** : brique essentielle qui représente la version 1 de spring. Version actuelle : 4.2
- **Spring Web Flow** : développement d'interfaces web riches (ajax, jsf,...), utilise Spring MVC. Version : 2.4
- **Spring Security** : sécurité au niveau d'une application JEE (authentification et habilitation des utilisateurs) Version Actuelle : 4

Galaxie Spring

modules d'intégration applicative



- **Spring Web Services** : version 2.2
- **Spring Batch** : plan de production pour l'enchaînement de traitements par lots liés par des dépendances, version 3
- **Spring OSGI** : architectures OSGi (Open Services Gateway initiative) – version 1.2
- **Spring Integration** : intégration dans un SI de l'entreprise (ESB ou squelette EAI) – version 4.1

Galaxie Spring

modules de développement



- **Spring ROO** : accompagner le processus de développement RAD, version 1.3
- **Spring IDE** : ensemble d'extensions pour des IDE (eclipse)
- **Spring STS** : Spring Source Tools Suite (environnement intégré basé sur eclipse). Version 3.6.4
- **Spring Bean Doc** : génération de documentation version 0.9

Galaxie Spring

modules de d'ouverture technologique



- **Spring Rich Client** : support d'application Swing, version 1.1 alpha – 06/2009
- **Spring .Net** : portage de Spring vers la plateforme .Net version 1.3 – 12/2009
- **Spring Blaze DS** : support d'applications Flex, version 1-03/2010
- **Spring Modules** : ANT, EHCache, Jboss Cache,... 03/2008
- **Spring Extensions** : développement d'add-ons Spring
- **Spring Java Config** : configuration par annotations – beta
- **Spring LDAP** : v1.3 – 08/2006 (pas d'évolutions depuis)
- ...

Développer avec Spring

- La formation portera sur la version 5 de Spring
- Tous les IDE supportant Java SE/Java EE :



NetBeans



IntelliJIDEA

à condition de récupérer les bibliothèques nécessaires (framework Spring, JMS, log, AOP, ...)

- Pivotal fournit un environnement complet
 - Un IDE basé sur Eclipse
 - Un serveur basé sur Tomcat



Configurer des beans Spring Core



Configuration de projet

- Créer un projet Maven de type Webapp
- Ajouter la dépendance spring-webmvc
- Par le jeu des dépendances liées, on a les dépendances suivantes :

```
▼ spring-webmvc : 5.0.4.RELEASE [compile]
  ▶ spring-aop : 5.0.4.RELEASE [compile]
  ▶ spring-beans : 5.0.4.RELEASE [compile]
  ▶ spring-context : 5.0.4.RELEASE [compile]
  ▶ spring-core : 5.0.4.RELEASE [compile]
  ▶ spring-expression : 5.0.4.RELEASE [compile]
  ▶ spring-web : 5.0.4.RELEASE [compile]
```

- Autres dépendances : junit, jstl, ...

Injection de dépendances

Namespaces de Spring



- Beans : paramétrage des objets
- Utils : faciliter le paramétrage des beans en XML, collections, ...
- Jee : accéder à des objets JEE (ressources JNDI, stateless bean)
- Lang : utilisation d'objets écrits en langage dynamique type Groovy
- Jms : utilisation de l'API de messaging Java
- Tx : gérer des transactions
- Aop : programmation par aspect de Spring
- Context : contexte de l'application (annotations, ...)
- Jdbc : configuration de datasources
- Cache : gestion de cache

Configuration de beans



- **Définition d'un JavaBean**

```
public class Contact {  
    private String nom;  
    private String prenom;  
    ...  
}
```

- **Définition d'un fichier XML de configuration et injection par mutateur**

```
<?xml version="1.0" encoding="UTF-8"?>  
<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-3.0.xsd">  
  
    <bean id="contact1" class="org.dawan.formation.Contact">  
        <property name="nom" value="DOE"></property>  
        <property name="prenom" value="John"></property>  
    </bean>  
  
</beans>
```

- **Récupération de l'instance du bean via le contexte d'application**

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");  
Contact c1 = (Contact) context.getBean("contact1");
```


Injection par constructeur



- Définition d'un ou plusieurs constructeurs

```
public class Contact {  
    private String nom;  
    private String prenom;  
    public Contact(String nom, String prenom) {  
        ...  
    }  
}
```

- Injection par constructeur

```
<bean id="contact1" class="org.dawan.formation.Contact">  
    <constructor-arg value="DOE"></constructor-arg>  
    <constructor-arg value="John"></constructor-arg>  
</bean>
```

Si il subsiste une ambiguïté dans les constructeurs, préciser index et type

Portée des beans (scope)



- **Prototype**

crée une instance à chaque demande (getBean)

- **Singleton**

Crée une instance unique pour chaque conteneur IoC

- **Request (web)**

Crée une instance par requête HTTP

- **Session**

Crée une instance par session HTTP

Contrôle des propriétés



Définition du type de vérification (dépréciée depuis Spring3)

```
<bean id="contact1" class="org.dawan.formation.Contact" dependency-check="...">  
  <property name="nom" value="DOE"></property>  
  <property name="prenom" value="John"></property>  
</bean>
```

Valeurs possibles :

- none
- Simple : vérification sur les types primitifs et les collections
- Objects : vérification sur les types autres que simple
- All : vérification de tous les types

NB : seules les propriétés injectées par mutateur sont contrôlées Une propriété initialisée dans un constructeur ne sera pas contrôlée.

Contrôle par annotation



Objectif : Pouvoir vérifier la définition d'une propriété plus finement

Dans le fichier de définition

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <context:annotation-config/>
    <bean id="contact1" class="org.dawan.formation.Contact">
        <constructor-arg value="DOE"></constructor-arg>
        <constructor-arg value="John"></constructor-arg>
    </bean>

</beans>
```

Dans le bean

```
@Required
public void setNom(String nom) {
    this.nom = nom;
}
```

Cycle de vie des beans



- **Interfaces InitializingBean et DisposableBean**

Fournissent les méthodes `afterPropertiesSet` et `destroy`

- **Annotations `@PostConstruct` et `@PreDestroy`**

Annotations JEE, caractérisent les méthodes appelées après la construction ou avant la destruction du bean

- **Attributs `init-method` et `destroy-method`**

Permet de préciser dans le fichier de configuration les méthodes d'initialisation et de destruction du bean

Héritage de configuration



- Définition d'un JavaBean

```
public class Client extends Contact {  
    private String numero;  
    ...  
}
```

- Définition dans le fichier de configuration

```
<bean id="contact1" class="org.dawan.formation.Contact">  
    <property name="nom" value="USER1"></property>  
    <property name="prenom" value="Mohamed"></property>  
</bean>  
<bean id="client1" class="org.dawan.formation.Client" parent="contact1">  
    <property name="numero" value="DA-123456"></property>  
</bean>
```

- Redéfinition d'attributs

```
<bean id="client1" class="org.dawan.formation.Client" parent="contact1">  
    <property name="nom" value="DERKAOUI"></property>  
    <property name="numero" value="DA-123456"></property>  
</bean>
```

Lier des beans (1)



Définition du mode de liaison dans le fichier de configuration

```
<bean id="contact1" class="org.dawan.formation.Contact" autowired="...">  
  <property name="nom" value="DOE"></property>  
  <property name="prenom" value="John"></property>  
</bean>
```

Valeurs possibles :

- no : aucune liaison automatique, on lie les beans dans le code
- byName: liaison entre le nom de la propriété et celui du bean
- byType : liaison entre le nom de la propriété et celui du bean
- constructor : liaison en utilisant le constructeur du bean
- autodetect : ~~Spring choisit entre byType et constructor (déprécié)~~

Lier des beans (2)



Définition de la liaison par annotation

```
// Liaison sur l'attribut
@Autowired
Private Address address;

// Liaison sur le constructeur
@Autowired
public Contact(String nom, String prenom, Address address) {
    ...
}

// Liaison sur le setter
@Autowired
public void setAddress(Address address) {
    this.address = address;
}
```

Spring va essayer de lier un bean dont le type est compatible.

Par défaut, les propriétés sont obligatoires.

Possibilité de spécifier le caractère facultatif

```
@Autowired(required=false)
```

Lier des beans (3)



Plusieurs beans correspondent ?

```
// Liaison sur l'attribut
@Autowired
@Qualifier("address1") // id du bean dans le fichier de config
Private Address address;

// Liaison lors du passage de paramètre
@Autowired
public inject(@Qualifier("address1") Address address) {
    ...
}

// Liaison sur un ensemble d'attributs
@Autowired
Private Address[] addresses;

Private List<Address> addresses;

// Indexés sur l'id
Private Map<String, Address> addresses;

// Liaison sur l'attribut
@Resource(name = "adress1") // id du bean
Private Address address;
```

Ajouter d'un
qualifieur

Gérer l'ensemble des
beans

Accéder au bean par
nom

Héritage de configuration



- Définition d'un JavaBean

```
public class Client extends Contact {  
    private String numero;  
    ...  
}
```

- Définition dans le fichier de configuration

```
<bean id="contact1" class="org.dawan.formation.Contact">  
    <property name="nom" value="DOE"></property>  
    <property name="prenom" value="John"></property>  
</bean>  
<bean id="client1" class="org.dawan.formation.Client" parent="contact1">  
    <property name="numero" value="DA-123456"></property>  
</bean>
```

- Redéfinition d'attributs

```
<bean id="client1" class="org.dawan.formation.Client" parent="contact1">  
    <property name="prenom" value="Jane"></property>  
    <property name="numero" value="DA-123456"></property>  
</bean>
```

Template de configuration



- **Héritage de configuration sans instantiation du parent**

```
<bean id="contact1" class="org.dawan.formation.Contact" abstract="true">
  <property name="nom" value="DOE"></property>
  <property name="prenom" value="John"></property>
</bean>
<bean id="client1" class="org.dawan.formation.Client" parent="contact1">
  <property name="numero" value="DA-123456"></property>
</bean>
```

- **Pas d'héritage Java mais caractéristiques communes**

```
<bean id="personne1" abstract="true">
  <property name="nom" value="DOE"></property>
  <property name="prenom" value="John"></property>
</bean>
<bean id="client1" class="org.dawan.formation.Client" parent="personne1">
  <property name="numero" value="DA-123456"></property>
</bean>
<bean id="formateur1" class="org.dawan.formation.Formateur" parent="personne1">
  <property name="matricule" value="JDOE0415"></property>
</bean>
```

Configuration de collections



```
public class Contact {  
    private String nom;  
    private String prenom;  
    ...  
    private List<Object> objects;  
    public void setObjects(List<Object> objects) {  
        this.objects = objects;  
    }  
    ...  
    private Object[] objects;  
    public void setObjects(Object[] objects) {  
        this.objects = objects;  
    }  
  
    private Set<Object> objects;  
    ...  
    public void setObjects(Set<Object> objects) {  
        this.objects = objects;  
    }  
}
```

```
<bean id="contact1" class="org.dawan.formation.Contact">  
    <constructor-arg value="MARRON"/>  
    <constructor-arg value="Benjamin"/>  
    <property name="objects">
```

```
        <list>  
            <value>A</value>  
            <bean class="java.net.URL">  
                <constructor-arg value="http"/>  
                <constructor-arg value="www.dawan.fr"/>  
                <constructor-arg value="/" />  
            </bean>  
            <null/>  
        </list>
```

```
        <set>  
            <value>A</value>  
            <bean class="java.net.URL">  
                <constructor-arg value="http"/>  
                <constructor-arg value="www.dawan.fr"/>  
                <constructor-arg value="/" />  
            </bean>  
            <null/>  
        </set>
```

```
    </property>  
</bean>
```

Configuration de collections



```
public class Contact {  
    private String nom;  
    private String prenom;  
    ...  
    private Map<Object,Object> objects;  
    public void  
    setObjects(Map<Object,Object> objects) {  
        this.objects = objects;  
    }  
}
```

```
<bean id="contact1" class="org.dawan.formation.Contact">  
    <constructor-arg value="MARRON"/>  
    <constructor-arg value="Benjamin"/>  
    <property name="objects">  
        <map>  
            <entry key="type" value="A"/>  
            <entry>  
                <value>URL</value>  
                <bean class="java.net.URL">  
                    <constructor-arg value="http"/>  
                    <constructor-arg value="www.dawan.fr"/>  
                    <constructor-arg value="/"/>  
                </bean>  
            </entry>  
        </map>  
    </property>  

```

```
private Properties myProps;  
public void setObjects(Properties objects) {  
    this.myProps = objects;  
}  
}
```

```
<property name="myProps">  
    <props>  
        <prop key="type">A</prop>  
        <prop key="URL">http://www.dawan.fr</prop>  
    </props>  
</property>  
</bean>
```

Collection et Héritage



- **Compléter une collection du bean parent**

```
<bean id="contact1" class="org.dawan.formation.Contact">
  <property name="nom" value="MARRON"/>
  <property name="prenom" value="Benjamin"/>
  <property name="objects">
    <list>
      <value>A</value>
      <value>B</value>
      <value>C</value>
    </list>
  </property>
</bean>
<bean id="client1" class="org.dawan.formation.Client" parent="contact1">
  <property name="numero" value="DA-123456"/>
  <property name="objects">
    <list merge="true">
      <value>A</value>
      <value>E</value>
      <value>D</value>
    </list>
  </property>
</bean>
```

- **Fonctionne pour tous les types de collections (list, set, map)**

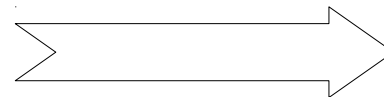
Typing les éléments

- Dans le fichier de configuration

```
<bean id="contact1" class="org.dawan.formation.Contact">
  <constructor-arg value="MARRON"/>
  <constructor-arg value="Benjamin"/>
  <property name="objects">
    <list>
      <value type="int">5</value>
      <value type="int">10</value>
      <value type="int">15</value>
    </list>
    <set type="int">
      <value>5</value>
      <value>10</value>
      <value>15</value>
    </set>
  </property>
</bean>
```

```
private List<Integer> objects;
• public void setObjects(List<Integer> objects) {
  this.objects = objects;
}
```

Dans le code



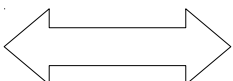
```
<list>
  <value>5</value>
  <value>10</value>
  <value>15</value>
</list>
```

Pas besoin de typer dans le fichier de configuration

Typing a collection

- **Spécifier la classe concrète de la collection**

```
<property name="objects">
  <bean class="org.springframework.beans.factory.config.SetFactoryBean">
    <property name="targetSetClass">
      <value>java.util.TreeSet</value>
    </property>
    <property name="sourceSet">
      <set>
        <value>5</value>
        <value>10</value>
        <value>20</value>
      </set>
    </property>
  </bean>
</property>
```



```
<property name="objects">
  <util:set set-class="java.util.TreeSet">
    <value>5</value>
    <value>10</value>
    <value>20</value>
  </util:set>
</property>
```

- **Réutiliser une collection**

```
<property name="objects">
  <ref local="objects"/>
</property>

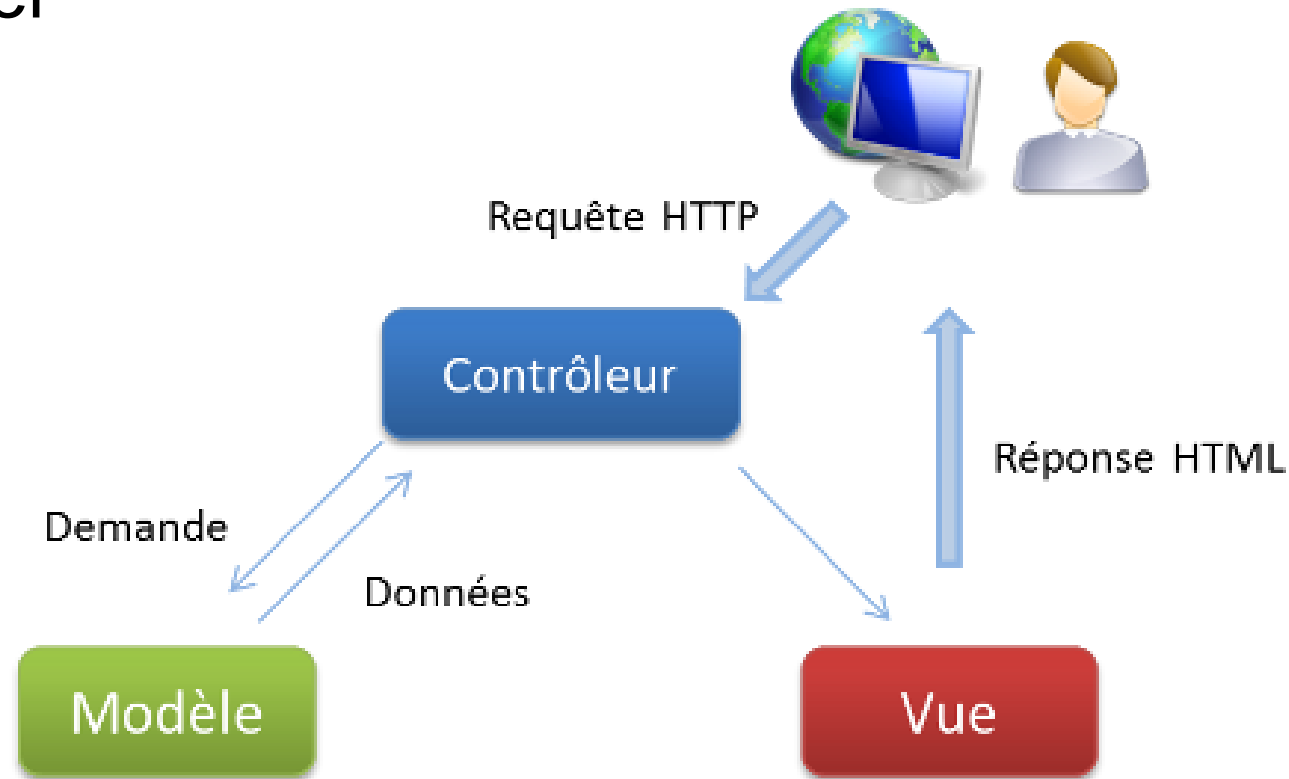
<util:set id="objects">
  <value>5</value>
  <value>10</value>
  <value>20</value>
</util:set>
```

Spring MVC



L'architecture MVC

Model-View-Controller : désigne la séparation des données, du traitement sur ces données et la manière de les restituer



L'architecture MVC2



MVC2 : introduction d'un *front controller* qui traite toutes les demandes et les renvoie au bon traitement.

- MVC2 n'est pas le successeur de MVC.
- MVC2 est plus complexe que MVC.
- MVC2 sépare la logique de la présentation contrairement à MVC.
- MVC2 est plus flexible que MVC.
- MVC2 est plus adapté pour de grosses applications.
- MVC correspond bien à de petites applications.

Frameworks MVC2

Orienté requête

 **Stripes**

 **spring**

 **Struts²**

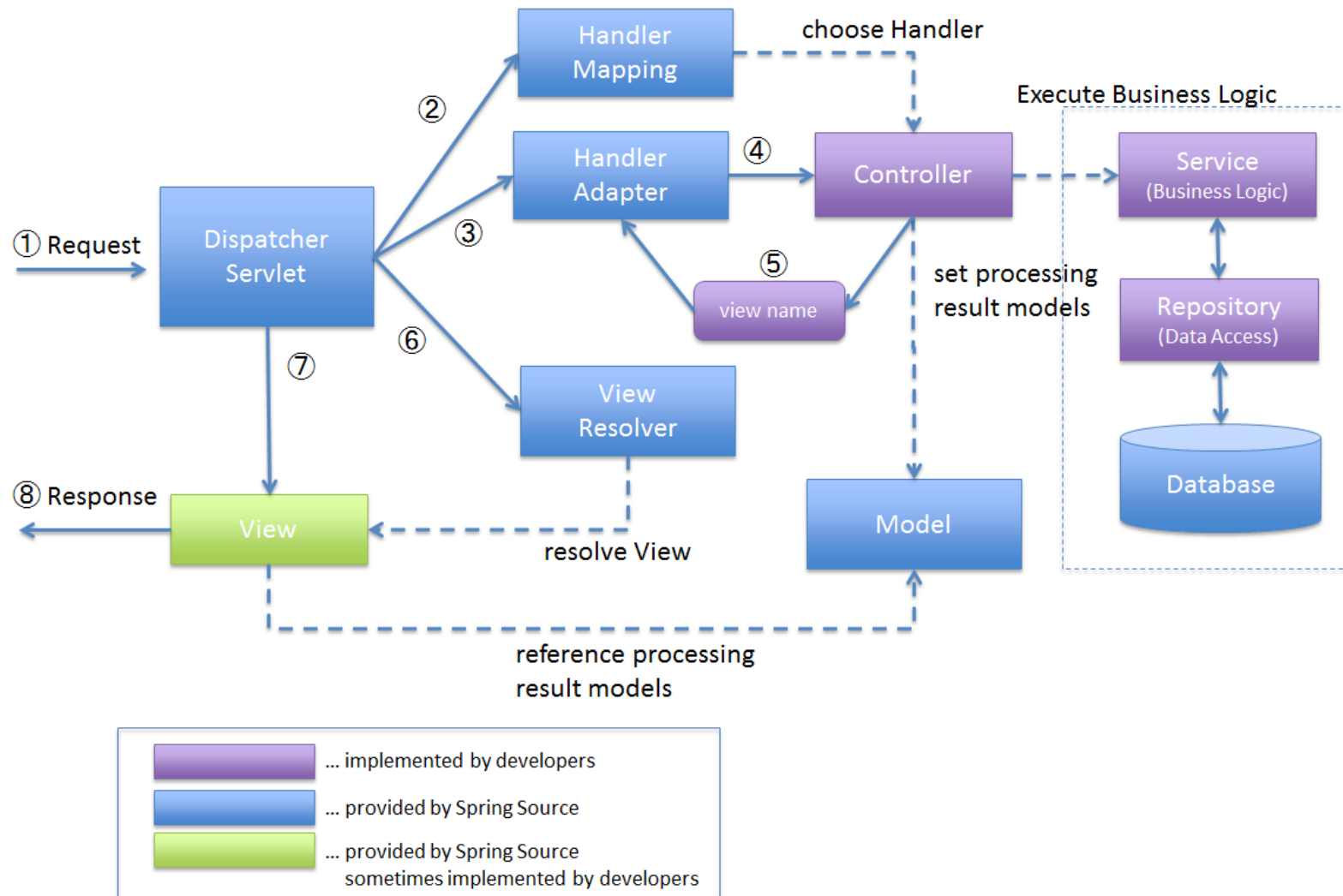
Orienté composant

 **JSF**

 **APACHE WICKET**

 **tapestry**

L'architecture MVC2 de Spring



<https://terasolunaorg.github.io/guideline/1.0.1.RELEASE/en/Overview/SpringMVCOverview.html>

Fonctionnement

- S'exécute dans un container léger : Tomcat
- Front Controller : servlet **DispatcherServlet**
- Contrôleurs : des POJO/JavaBean annotés @Controller
- Vues : choix possible de la technologie jsp (ou Tapestry, Struts Tiles)
- Un mapping request dans le contrôleur : @RequestMapping
- Des objets métiers : Objets Spring ou objets JEE

Configuration de l'application via XML



Configurer le contrôleur principal dans le web.xml

```
<web-app id="WebApp_ID" version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>root</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/appServlet/root-servlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>root</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```



Attention à la version de JSP : préférer la version 2.0 de JSP au lieu de la 1.2

Configuration de l'application via XML (2)



Configurer Spring dans le fichier de contexte Web : root-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:context="http://www.springframework.org/schema/context"
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.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

<mvc:annotation-driven/>
<context:component-scan base-package="fr.dawan.controller"/>
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/jsp/" />
  <property name="suffix" value=".jsp" />
</bean>
</beans>
```

Configuration de l'application via XML (3)



```
<mvc:annotation-driven/>
```

- ◆ Indique que les annotations seront utilisées dans le code Java

```
<context:component-scan base-package="fr.dawan"/>
```

- ◆ Précise le package dans lequel seront contenus les classes devant être découvertes par Spring

```
<bean id="viewResolver"  
class="org.springframework.web.servlet.view.InternalResourceViewResolver"  
>  
    <property name="prefix" value="/WEB-INF/jsp/" />  
    <property name="suffix" value=".jsp" />  
</bean>
```

- ◆ Définit la méthode de détermination des vues à utiliser

Configuration de l'application via annotations



Configurer l'application : équivalent du contenu du web.xml

La définition et le paramétrage de la DispatcherServlet est pris en charge par la super classe.

```
public class ApplicationInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { ApplicationConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

Configuration de l'application via annotations (2)



Configurer Spring : équivalent du fichier root-context.xml

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages= {"fr.dawan"})
public class ApplicationConfig implements WebMvcConfigurer{

    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.jsp("/WEB-INF/jsp/", ".jsp");
    }
}
```

- ◆ @Configuration : indique qu'il s'agit d'une classe de configuration
- ◆ @ComponentScan : répertoire contenant les composants de l'application
- ◆ configureViewResolvers : ajout du mapping pour les JSP

Contrôleur



Contrôleur Spring



Classe Java annotée @Controller

- Contient des méthodes liées à des mapping de requêtes
 - ♦ @RequestMapping
- Mappings spécialisés pour les méthodes HTTP :
 - ♦ @GetMapping
 - ♦ @PostMapping
 - ♦ @PutMapping
 - ♦ @DeleteMapping
 - ♦ @PatchMapping

Contrôleur Spring



```
@Controller
public class WelcomeController {

    @RequestMapping("/home")
    public String home() {
        return "home";
    }

    @GetMapping("/greeting")
    public String greetingByGet() {
        return "greetingGet";
    }

    @PostMapping("/greeting")
    public String greetingByPost() {
        return "greeting";
    }
}
```

RequestMapping

Peut être appliqué sur une méthode ou un contrôleur

- URL :

- ♦ `@RequestMapping("/users")`
- ♦ `@RequestMapping("/users", "/clients")`

- URI templates :

- ♦ `@RequestMapping("/users/{userId}")`
- ♦ `@RequestMapping("/users/{userId:[0-9]++}")`

- Méthodes HTTP :

- ♦ `@RequestMapping(method={RequestMethod.GET})`
- ♦ `@RequestMapping(method={RequestMethod.GET, ...})`

RequestMapping (2)

- Paramètres :

- ♦ `@RequestMapping(params="id=8")`
- ♦ `@RequestMapping(params={"id=8", "name=DOE"})`
- ♦ `@RequestMapping(params="id")`

- Entêtes :

- ♦ `@RequestMapping(headers="host=127.0.0.1")`

- Consommation/production :

- ♦ `@RequestMapping(consumes=MediaType.APPLICATION_JSON_VALUE)`
- ♦ `@RequestMapping(produces=MediaType.APPLICATION_JSON_VALUE)`

Les paramètres de l'annotation peuvent être combinés entre eux.

PathVariable

Identifier un élément de l'URI

- Variable nommée

```
@RequestMapping("/users/{id}")  
public String handleRequest (@PathVariable("id") String userId, Model map) {  
    return "my-page";  
}
```

- Variable avec nommage implicite

```
@RequestMapping("/users/{id}")  
public String handleRequest (String id, Model map) {  
    return "my-page";  
}
```

- Variables multiples

```
@RequestMapping("/users/{id}/adress/{adrId}")  
public String handleRequest (@PathVariable ("id") String userId, @PathVariable("id") String  
    adrId, Model map) {  
    return "my-page";  
}
```

PathVariable (2)

- Variables multiples dans une map

```
@RequestMapping("/users/{id}/address/{adrId}")
public String handleRequest (@PathVariable Map<String, String> varsMap, Model map) {
    return "my-page";
}
```

- Variables ambiguës

```
@RequestMapping("/users/{id}")
public String handleRequest (@PathVariable("id") String userId, Model model){
    return "my-page";
}
```

```
@RequestMapping("/users/{name}")
public String handleRequest2 (@PathVariable("name") String userName, Model model) {
    return "my-page";
}
```

→ Solution : introduire des expressions régulières

```
@RequestMapping("/users/{id:[0-9]+}")
public String handleRequest (@PathVariable("id") String userId, Model model){...}

@RequestMapping("/users/{name:[A-Za-z]+}")
public String handleRequest2 (@PathVariable("name") String userName, Model model) {...}
```

RequestParam

Identifier un paramètre de l'URL

- Paramètre nommé

```
@RequestMapping
public String handleEmployeeRequestByDept (@RequestParam("dept") String deptName, Model map) {
    return "my-page";
}
```

- Paramètre avec nommage implicite

```
@RequestMapping
public String handleEmployeeRequestByDept (@RequestParam String dept, Model map) {
    return "my-page";
}
```

- Paramètres multiples

```
@RequestMapping
public String handleEmployeeRequestByDept (@RequestParam("dept") String deptName,
                                           @RequestParam("state") String stateCode, Model map) {
    return "my-page";
}
```

RequestParam (2)

- Paramètres multiples dans une map

```
@RequestMapping()  
public String handleRequest (@RequestParam Map<String, String> paramsMap, Model map)  
{  
    return "my-page";  
}
```

- Paramètres ambigus : params dans le RequestMapping

```
@RequestMapping(params = "dept")  
public String handleEmployeeRequestByDept (@RequestParam("dept") String deptName, Model map) {  
    return "my-page";  
}  
  
@RequestMapping(params = "state")  
public String handleEmployeesRequestByArea (@RequestParam("state") String state, Model map) {  
    return "my-page";  
}
```

- Paramètre requis, valeur par défaut

```
@RequestMapping("/users")  
public String handleRequest(@RequestParam(value = "project", defaultValue="kara") String projectName,  
                           Model model){  
    return "my-page";  
}
```


RequestParam (3)

- Conversion implicite

```
@RequestMapping()  
public String handleRequest (@RequestParam("nbitems") int nbItems, Model model) {  
    return "my-page";  
}
```

- Conversion de format de dates

```
@RequestMapping()  
public String handleRequest (@PathVariable("id") String employeeId,  
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)  
    @RequestParam("startDate") LocalDate startDate,  
    Model model) {  
    return "my-page";  
}
```

RequestHeader

Propose les mêmes fonctionnalités que RequestParam

- Entête nommé
- Entête avec nommage implicite
- Entêtes multiples
- Entêtes stockés dans une map
- Entête requis, valeur par défaut
- Conversion de type

+ Récupération de tous les entêtes :

```
@RequestMapping
public String handleRequestWithAllHeaders (@RequestHeader HttpHeaders httpHeaders, Model model) {
    return "my-page";
}
```

Lier des beans

Objectif : pouvoir mettre des types complexes dans les signatures des méthodes du contrôleur

- Création d'un JavaBean

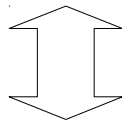
```
public class User implements Serializable{  
    private int id;  
    private String firstName;  
    private String lastName;  
  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
    ...  
}
```

Lier des beans (2)

- Utiliser les paramètres de la requête

```
@Controller
@RequestMapping("users")
public class UserController {

    @RequestMapping
    public String handleTradeRequest(
        @RequestParam("id") String id,
        @RequestParam("firstName") String firstName,
        @RequestParam("lastName") String lastName,
        Model map) {
        String msg =
            String.format("User: %s %s %s"
                , id, firstName, lastName);
        return "my-page";
    }
}
```



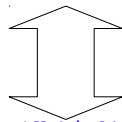
```
@RequestMapping
public String handleTradeRequest (User user, Model map) {
    String msg = String.format("User: %s %s %s", user.getId(),
        user.getFirstName(), user.getLastName());
    return "my-page";
}
}
```

Lier des beans (3)

- Utiliser les composants de l'URL

```
@Controller
@RequestMapping("users")
public class UserController {

    @RequestMapping("{id}/{lastName}/{firstName}")
    public String handleTradeRequest(
        @PathVariable("id") String id,
        @PathVariable("firstName") String firstName,
        @PathVariable("lastName") String lastName,
        Model map) {
        String msg =
            String.format("User: %s %s %s",
                id, firstName, lastName);
        return "my-page";
    }
}
```



```
@RequestMapping("{id}/{lastName}/{firstName}")
public String handleTradeRequest (User user, Model map) {
    String msg = String.format("User: %s %s %s", user.getId(),
        user.getFirstName(), user.getLastName());
    return "my-page";
}
```

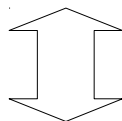
```
}
```

Lier des beans (3)

- Utiliser les composants de l'URL

```
@Controller
@RequestMapping("users")
public class UserController {

    @RequestMapping("{id}/{lastName}/{firstName}")
    public String handleTradeRequest(
        @PathVariable("id") String id,
        @PathVariable("firstName") String firstName,
        @PathVariable("lastName") String lastName,
        Model map) {
        String msg =
            String.format("User: %s %s %s",
                id, firstName, lastName);
        return "my-page";
    }
}
```



```
@RequestMapping("{id}/{lastName}/{firstName}")
public String handleTradeRequest (User user, Model map) {
    String msg = String.format("User: %s %s %s", user.getId(),
        user.getFirstName(), user.getLastName());
    return "my-page";
}
```

```
}
```

ModelAttribute

Permet d'annoter une méthode qui retourne la valeur d'une entrée du modèle

```
@Controller
public class WelcomeController {

    @GetMapping("/greeting")
    public String greetingByPost(Model model) {
        model.addAttribute("titre", "Test de vue");
        model.addAttribute("description", "pour tester le passage d'infos du controleur");
        return "default";
    }

    @ModelAttribute("current_time")
    public LocalDateTime getCurrentTime() {
        return LocalDateTime.now();
    }

    @ModelAttribute("user")
    public void getRequestingUser(@RequestParam(value="name", required=false) String userName,
                                  Model model) {
        model.addAttribute("user", userName==null?"John DOE":userName);
    }
}
```

Fonctionne aussi avec @PathVariable

ModelAttribute (2)

Utiliser en paramètre d'une méthode du contrôleur

```
@Controller
public class WelcomeController {

    @GetMapping("/greeting")
    public String greetingByPost(@ModelAttribute("user") User user, Model model) {
        model.addAttribute("titre", "Test de vue");
        model.addAttribute("description", "pour tester le passage d'infos du controleur");
        if (user.getLastName().equals("DOE")) {
            return "default";
        } else {
            return "greeting";
        }
    }

    @ModelAttribute("user")
    public User getRequestingUser(@RequestParam(value="name", required=false) String userName,
    Model model) {
        User u = new User();
        u.setLastName(userName==null?"DOE":userName);
        return u;
    }
}
```


Stockage en session

Stocker des informations en session Http via @SessionAttributes

```
@Controller
@SessionAttributes("user")
public class WelcomeController {

    ...

    @ModelAttribute("user")
    public User getRequestingUser(@RequestParam(value="name", required=false) String userName,
    Model model) {
        User u = new User();
        u.setLastName(userName==null?"DOE":userName);
        return u;
    }
}
```

- ◆ Recherche de user dans la session Http
- ◆ Si non trouvé, appel de la méthode @ModelAttribute et stockage du résultat dans la session

Stockage en session (2)

Utiliser l'injection de dépendance de Spring

- Classe de configuration : injecter l'objet en session

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages= {"fr.dawan"})
public class ApplicationConfig implements WebMvcConfigurer{

    @Bean
    @Scope(WebApplicationContext.SCOPE_SESSION)
    public User user(HttpServletRequest request){
        return new User();
    }
}
```

- Dépendances : spring-web

Stockage en session (3)

Utiliser l'injection de dépendance de Spring

- Contrôleur : utiliser un Provider pour accéder à l'objet

```
@Controller
@SessionAttributes("user")
public class WelcomeController {

    @Autowired
    private Provider<User> userProvider;

    @GetMapping("/greeting")
    public String greetingByPost(Model model) {
        User user = userProvider.get();
        if (user.getLastName().equals("DOE")) {
            return "default";
        } else {
            return "greeting";
        }
    }
}
```

- Dépendances : javax.inject

Intercepteur

- 3 méthodes :
 - ◆ PreHandle : avant la méthode du controleur
 - ◆ PostHandle : entre la méthode du controleur et la vue
 - ◆ AfterCompletion : une fois que la vue est rendue

```
public class MyCounterInterceptor extends HandlerInterceptorAdapter {  
    private AtomicInteger counter = new AtomicInteger(0);  
  
    @Override  
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object  
handler) throws Exception {  
        HttpSession session = request.getSession(true);  
        request.setAttribute("visitorCounter", counter.incrementAndGet());  
        return true;  
    }  
}
```

- Accès à la session et à la requête

Intercepteur (2)

- Contrôleur : en paramètre de méthode

- ◆ Accès à la session

```
@SessionAttribute(name = "sessionStartTime")
```

- ◆ Accès à la requête

```
@RequestAttribute(name = "sessionStartTime")
```

Cookies



- Accès en lecture : en paramètre de méthode

```
@CookieValue(value = "myCookieName", defaultValue = "defaultCookieValue")
```

- Accès en lecture
 - ♦ Récupération de la `HttpRequest` en la mettant en paramètre de méthode
 - ♦ Lecture du cookie standard JEE
- Accès en écriture
 - ♦ Récupération de la `HttpResponse` en la mettant en paramètre de méthode
 - ♦ Écriture du cookie standard JEE

Gestion des exceptions

- Annotation d'une méthode `@ExceptionHandler` dans le contrôleur :
 - ◆ Cette méthode sera invoquée lors du lancement d'une exception

```
@ExceptionHandler
public String handleError() {...}
```
 - ◆ Possibilité de spécifier l'exception à gérer

```
@ExceptionHandler({SQLException.class, DataAccessException.class})
public String databaseError() {...}
```
 - ◆ Cascade de gestionnaires d'exception
- Généraliser la gestion des exceptions dans une classe annotée `@ControllerAdvice`

Vues



Types de vues

Spring permet d'utiliser différents types de vues :

- JSP/JSTL
- Thymeleaf
- FreeMarker
- Groovy Markup
- Tiles
- XML, CSV, JSON

Liaison vues-contrôleur



Utilisation d'un modèle contenant des attributs :

```
@Controller
public class WelcomeController {

    @RequestMapping("/home")
    public String home() {
        return "home";
    }

    @GetMapping("/greeting")
    public String greetingByGet() {
        return "greetingGet";
    }

    @PostMapping("/greeting")
    public String greetingByPost(Model model) {
        model.addAttribute("titre", "Test de vue");
        model.addAttribute("description", "pour tester le passage d'infos du controleur");
        return "default";
    }
}
```

Définition d'une vue JSP

Utilisation d'un ViewResolver pour déterminer le type et l'emplacement du fichier

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>${titre}</title>
  </head>
  <body>
    <h2>${titre}</h2>
    <p>${description}</p>
  </body>
</html>
```



Attention à la version de JSP : si version 1.2, ajouter dans la balise head : `<%@ page isELIgnored="false" %>`

Spring Form Taglib

- Spring propose une Taglib JSP pour faciliter la gestion des formulaires
 - ◆ Balises form, button, checkbox ... pour les différents éléments
 - ◆ Balise errors et attribut cssErrorClass pour la gestion des erreurs
 - ◆ Attributs de gestion d'événements sur les éléments : onclick, ondoubleclick, onkeyup, onkeydown, onmouseup, onmouseover, onmousedown

Spring Form Taglib (2)

Tag
<form>
<input>
<hidden>
<checkbox>
<checkboxes>
<radiobutton>
<radiobuttons>
<select>
<option>
<options>
<errors>
<label>
<password>
<textarea>
<button>

```
<form:form method="POST" commandName="user">
  <table>
    <tr>
      <td>User Name :</td>
      <td>
        <form:input path="name" />
      </td>
    </tr>
    <tr>
      <td>Password :</td>
      <td><form:password path="password" /></td>
    </tr>
    <tr>
      <td>Gender :</td>
      <td>
        <form:radiobutton path="gender" value="M" label="M" />
        <form:radiobutton path="gender" value="F" label="F" />
      </td>
    </tr>
    <tr>
      <td>Country :</td>
      <td>
        <form:select path="country">
          <form:option value="0" label="Select" />
          <form:options items="${countryList}" itemValue="countryId"
            itemLabel="countryName" />
        </form:select>
      </td>
    </tr>
    <tr>
      <td colspan="2"><input type="submit" value="Register"></td>
    </tr>
  </table>
</form:form>
```

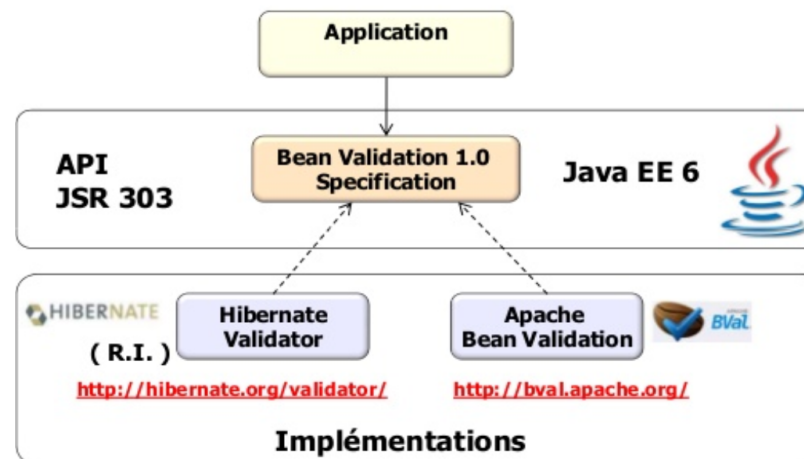
Gestion des formulaires



- Utilisation du binding automatique entre les attributs de la requête et les champs d'un Bean
 - ◆ Les champs du formulaire portent les mêmes noms que les propriétés de l'objet
 - ◆ Une instance de l'objet est mise en paramètre de la méthode du contrôleur
 - ◆ L'objet est automatiquement peuplé avec les données du formulaire

Bean Validation

- Framework standard de validation des données à différents niveaux : présentation, métier ou accès aux données.
- Implémentation de référence : **Hibernate Validator**
- Définit des annotations pour appliquer des contraintes, les messages d'erreur peuvent être gérés dans l'annotation.
- Apport de JEE 6 : Bean Validation 1.0 (JSR 303) : 2009 (package javax.validation.*), API mise à jour JEE 7 (v1.1 – JSR 349)



Apports de JEE 7

Bean Validation 1.1



- **Application de contraintes sur des paramètres de méthodes et des valeurs de retour.**
- Contraintes sur des constructeurs.
- Nouvelle API pour obtenir des meta-données de contraintes et les objets associés.
2 nouveaux packages :
`javax.validation.constraintvalidation`
`javax.validation.executable`
- **Meilleure intégration avec : JPA, CDI, JAX-RS, JSF,...**

Annotations

Annotation	Applicable aux types...
@Null @NotNull	Object
@Min @Max	BigDecimal, BigInteger, byte, short, int, long (+ Wrappers)
@DecimalMin @DecimalMax	BigDecimal, BigInteger, String byte, short, int, long (+ Wrappers)
@Size	String, Collection, Map, Array
@Digits	BigDecimal, BigInteger, String byte, short, int, long (+ Wrappers)
@Past @Future	java.util.Date, java.util.Calendar
@Pattern	String
@AssertTrue @AssertFalse	Boolean, boolean

Exemple

```
public class Book {  
    @NotNull  
    @Pattern(regexp = "^(97(8|9))?\d{9}(\\d|X)$")  
    private String isbn;  
  
    @NotNull  
    @Size(min = 1)  
    private List<String> authors;  
  
    @NotNull  
    @Size(min = 10)  
    private String title;  
  
    @Min(50)  
    private int pages;  
  
    @NotNull  
    @Size(min = 5)  
    private String publisher;  
}
```

- En cas d'agrégation d'objets, il faut annoter l'objet interne **@Valid**.

Validation des formulaires



- Utilisation de la JSR BeanValidation pour annoter l'objet à valider
- Indiquer dans la méthode du contrôleur que l'objet en paramètre doit être validé : utilisation de `@Valid`
- Récupérer les résultats de la validation dans un objet `BindingResult`

```
@RequestMapping(method = RequestMethod.POST)  
public String handlePostRequest (@Valid User user, BindingResult bindingResult,  
                                Model model) {  
    if (bindingResult.hasErrors()) {  
        ...  
        return "user-registration";  
    }  
  
    userService.saveUser(user);  
    return "registration-done";  
}
```

- Dépendance : hibernate-validator
- Le `BindingResult` doit obligatoirement suivre le bean à valider

Messages d'erreurs

- Définition dans les annotations de contraintes.
- Messages par défaut dans un fichier ValidationMessages.properties (voir le jar de l'implémentation).
Exemple (fichier de Hibernate Validator) :

```
javax.validation.constraints.AssertFalse.message = must be false
javax.validation.constraints.AssertTrue.message  = must be true
javax.validation.constraints.DecimalMax.message  = must be less than or equal to {value}
javax.validation.constraints.DecimalMin.message  = must be greater than or equal to {value}
javax.validation.constraints.Digits.message      = numeric value out of bounds
                                                    (<{integer} digits>.<{fraction} digits> expected)
javax.validation.constraints.Future.message      = must be in the future
javax.validation.constraints.Max.message         = must be less than or equal to {value}
javax.validation.constraints.Min.message         = must be greater than or equal to {value}
javax.validation.constraints.Past.message        = must be in the past
javax.validation.constraints.Pattern.message     = must match "{regex}"
javax.validation.constraints.Size.message        = size must be between {min} and {max}
```

Fichier
ValidationMessages.properties

- org.hibernate.validator
 - HibernateValidator.class
 - HibernateValidatorConfiguration.class
 - HibernateValidatorContext.class
 - HibernateValidatorFactory.class
 - ValidationMessages_cs.properties
 - ValidationMessages_de.properties
 - ValidationMessages_en.properties
 - ValidationMessages_es.properties
 - ValidationMessages_fr.properties
 - ValidationMessages_hu.properties
 - ValidationMessages_min_MN.properties
 - ValidationMessages_pt_BR.properties
 - ValidationMessages_tr.properties
 - ValidationMessages_zh_CN.properties
 - ValidationMessages.properties

Messages d'erreurs (2)

- Dans le bean

```
public class User {  
    private Long id;  
  
    @Size(min = 5, max = 20, message = "{user.name.size}")  
    private String name;  
  
    @Size(min = 6, max = 15, message = "{user.password.size}")  
    @Pattern(regexp = "\\S+", message = "{user.password.pattern}")  
    private String password;  
  
    @NotEmpty(message = "{user.email.empty}")  
    @Email(message = "{user.email.valid}")  
    private String emailAddress;  
}
```

- Dans la configuration

```
@Bean  
public MessageSource messageSource() {  
    ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();  
    messageSource.setBasenames("ValidationMessages");  
    return messageSource;  
}
```

Messages d'erreurs (3)

- Dans la JSP

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@taglib uri="http://www.springframework.org/tags/form" prefix="frm"%>
<html>
  <head>
  </head>
  <body>
    <h3>Registration Form</h3>
    <br />
    <frm:form action="register" method="post" commandName="user">
      <pre>
        Name <frm:input path="name" />
          <frm:errors path="name" cssClass="error" />
        Email address <frm:input path="emailAddress" />
          <frm:errors path="emailAddress" cssClass="error" />
        <input type="submit" value="Submit" />
      </pre>
    </frm:form>
  </body>
</html>
```

Spring Validator

- Implémenter l'interface Validator

```
public class UserValidator implements Validator {  
  
    public boolean supports(Class<?> clazz) {  
        return clazz == User.class;  
    }  
  
    public void validate(Object target, Errors errors) {  
        ValidationUtils.rejectIfEmpty(errors, "lastName", "user.name.empty");  
  
        User user = (User) target;  
        if (user.getLastName() != null && user.getLastName().length() < 5 ||  
            user.getLastName().length() > 20) {  
            errors.rejectValue("name", "user.name.size");  
        }  
    }  
}
```

- Dans le contrôleur

```
@RequestMapping("/{id}/{lastName}/{firstName}")  
public String handleRequest (User user, BindingResult bindingResult, Model map) {  
    new UserValidator().validate(user, bindingResult);  
    if (bindingResult.hasErrors()) {  
        ...  
    }  
    return "my-page";  
}
```

Internationalisation i18n

- 3 beans :
 - ◆ MessageSource : gérer les fichiers de messages
 - ◆ CookieLocaleResolver : stocker la locale
 - ◆ LocaleChangeInterceptor : gérer le changement de langue

```
<!-- définition de l'emplacements des fichiers de messages -->  
<beans:bean id="messageSource"  
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">  
<beans:property name="basename" value="classpath:messages" />  
<beans:property name="defaultEncoding" value="utf-8" />  
</beans:bean>
```

```
<!-- définition de la locale par défaut -->  
<beans:bean id="localeResolver"  
class="org.springframework.web.servlet.i18n.CookieLocaleResolver">  
<beans:property name="defaultLocale" value="fr" />  
</beans:bean>
```

```
<interceptors>  
<beans:bean id="localeChangeInterceptor"  
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">  
<beans:property name="paramName" value="lang" />  
</beans:bean>  
</interceptors>
```


Internationalisation (2)



- Dans le contrôleur :

```
@Autowired
private MessageSource messageSource;

@RequestMapping(value = "/", method = RequestMethod.GET)
public String handleRequest(Locale locale, Model model){

    // add parametrized message from controller
    String welcome = messageSource.getMessage("welcome.message", new Object[]{"John Doe"}, locale);
    model.addAttribute("message", welcome);

    // obtain locale from LocaleContextHolder
    Locale currentLocale = LocaleContextHolder.getLocale();
    model.addAttribute("locale", currentLocale);

    return "index";
}
```

L'utilisation de la locale en paramètre ou via LocaleContextHolder renvoie la même valeur : celle changée par l'intercepteur

Upload de fichiers

- Besoin d'un MultipartResolver dans la configuration

```
<beans:bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- max upload size in bytes -->
    <beans:property name="maxUploadSize" value="20971520" /> <!-- 20MB -->
    <!-- max size of file in memory (in bytes) -->
    <beans:property name="maxInMemorySize" value="1048576" /> <!-- 1MB -->
</beans:bean>
```

- Utilisation d'un MultipartFile dans le contrôleur

```
@RequestMapping(method = RequestMethod.POST)
public String handlePost(@RequestParam("user-file") MultipartFile multipartFile, Model model)
throws IOException {
    String name = multipartFile.getOriginalFilename();
    BufferedWriter w = Files.newBufferedWriter(Paths.get("d:\\filesUploaded\\" + name));
    w.write(new String(multipartFile.getBytes()));
    w.flush();

    model.addAttribute("msg", "File has been uploaded: " + name);
    return "response";
}
```

- Dépendance : commons-fileupload

Download de fichiers

- Ecriture directe dans la réponse HttpServletResponse au format choi
- Association d'un type MIME et d'un nom de fichier dans les entêtes de la réponse

```
@RequestMapping("/export-contacts")
public void generateCsv(HttpServletResponse response) {
    response.setContentType("text/csv");
    response.setHeader("Content-Disposition", "attachment;filename=contacts.csv");
    List<Contact> lc = ContactDao.findAll();
    try {

        ServletOutputStream out = response.getOutputStream();
        out.write(("id;name\n").getBytes()); //ligne d'entetes
        for (Contact cx : lc) {
            out.write((cx.getId() + ";" + cx.getName() + "\n").getBytes());
        }
        out.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Theming

- Paramétrer un ThemeResolver dans la configuration

```
<beans:bean id="themeResolver"  
class="org.springframework.web.servlet.theme.CookieThemeResolver">  
    <beans:property name="defaultThemeName" value="default" />  
</beans:bean>
```

- Créer des fichiers de propriétés se rapportant aux différents styles : CSS, images, ...
- Définir un intercepteur dans la configuration

```
<interceptors>  
    <beans:bean id="themeChangeInterceptor"  
class="org.springframework.web.servlet.theme.ThemeChangeInterceptor">  
        <beans:property name="paramName" value="theme" />  
    </beans:bean>  
</interceptors>
```

- Laisser la magie opérer ...

Templating

Complément de JSP

- SiteMesh

<http://wiki.sitemesh.org/wiki/display/sitemesh3/Getting+Started+with+SiteMesh+3>

Alternatives à JSP

- Thymeleaf

<https://www.thymeleaf.org/>

- FreeMarker

<https://freemarker.apache.org/>

- Velocity

https://fr.wikipedia.org/wiki/Apache_Velocity

Templating (2)

Exemple FreeMarker

- Dépendances : freemarker, spring-context-support
- Configurer Freemarker : extension des vues : .ftl

```
@Override
public void configureViewResolvers (ViewResolverRegistry registry) {
    registry.freeMarker();
}
@Bean
public FreeMarkerConfigurer freeMarkerConfigurer() {
    FreeMarkerConfigurer configurator = new FreeMarkerConfigurer();
    configurator.setTemplateLoaderPath("/WEB-INF/views/");
    return configurator;
}
```

- Contrôleur : rien ne change, une méthode retourne une vue

```
@RequestMapping
public String handleRequest (Model model) {
    model.addAttribute("msg", "A message from the controller");
    model.addAttribute("time", LocalTime.now());
    return "my-page";
}
```

Templating (3)

Exemple Template FreeMarker

```
<!DOCTYPE html>
<html lang="en">
<body>
<h2>FreeMarker View</h2>
  <div> Message: ${msg}</div>
  <div> Time: ${time} </div>
</body>
</html>
```



```
<!DOCTYPE html>
<html lang="en">
<body>
<h2>FreeMarker View</h2>
  <div> Message: A message from the controller</div>
  <div> Time: 22:39:13.512 </div>
</body>
</html>
```

Architecture REST



Architecture REST



Qu'est ce que c'est REST ?

- Representational State Transfer
- Repose sur l'architecture originelle du Web
- Utilise uniquement HTTP et un ensemble restreint d'actions : GET, POST, PUT, DELETE
- Utilise une URI (Uniform Resource Identifier) comme moyen d'interroger le service
- Des types MIME pour indiquer la nature des informations retournées par le service

Architecture REST (2)



Selon son créateur, Roy Fielding, une architecture REST doit respecter :

- Client-serveur : on sépare le producteur du consommateur pour garantir un faible couplage
- Sans état : une requête doit contenir l'ensemble des informations nécessaires au serveur pour traiter la demande
- Mise en cache : la réponse serveur peut contenir des informations comme la date de création ou de validité de la réponse
- Système par couches : on accède à des ressources individuelles, une fonctionnalité complexe entraîne un ensemble d'appel au serveur

Contrôleur REST

- Annoter le contrôleur avec `@RestController`
 - ◆ Évite de spécifier `@ResponseBody` sur chacune des méthodes du contrôleur
 - ◆ Format de la réponse : données au format JSON, XML possible
- Dépendance : jackson-databind
 - ◆ Conversion automatique des données du bean au format JSON

Contrôleur REST (2)

- Lecture
 - ♦ @GetMapping
- Création
 - ♦ @PostMapping
 - ♦ Données dans @RequestBody
- Mise à jour
 - ♦ @PutMapping
 - ♦ Données dans @RequestBody
 - ♦ Id en PathVariable
- Suppression
 - ♦ @DeleteMapping
 - ♦ Id en PathVariable

Utilisation de XML

- Annoter le bean avec JAXB

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.PROPERTY)
public class User implements Serializable{
    private int id;
    private String firstName;
    private String lastName;

    @XmlAttribute
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
}
```

- Indiquer au contrôleur que la méthode produit du XML

Exemple de contrôleur REST



```
@RestController
@RequestMapping("trainings") //@Path
public class WSTrainingController {

    @Autowired
    private TrainingDao trainingDao;

    @GetMapping(value = "/json", produces = "application/json")
    public List<Training> listAll() {
        return trainingDao.findAllBasicInfosTrainings();
    }

    @GetMapping(value = "/xml", produces = "application/xml")
    public List<Training> listAllXml() {
        return trainingDao.findAllBasicInfosTrainings();
    }

    @GetMapping(value =("/{id}", produces = "application/json")
    public Training find(@PathVariable int id) {
        return trainingDao.findBasicInfosTrainingById(id);
    }

    @PostMapping(value = "/insert", consumes = "application/json")
    public void find(@RequestBody Training training) {
        trainingDao.persist(training);
    }
}
```

Codes retour HTTP

- Annoter la méthode du contrôleur avec @ResponseStatus
 - ♦ Classe `HttpStatus`

Modèle : Spring ORM



Correspondance des modèles

« Relationnel - Objet »



- Le modèle objet propose plus de fonctionnalités :
 - L'héritage, le polymorphisme
- Les relations entre deux entités sont différentes
- Les objets ne possèdent pas d'identifiant unique contrairement au modèle relationnel

Accès aux Bdds en Java

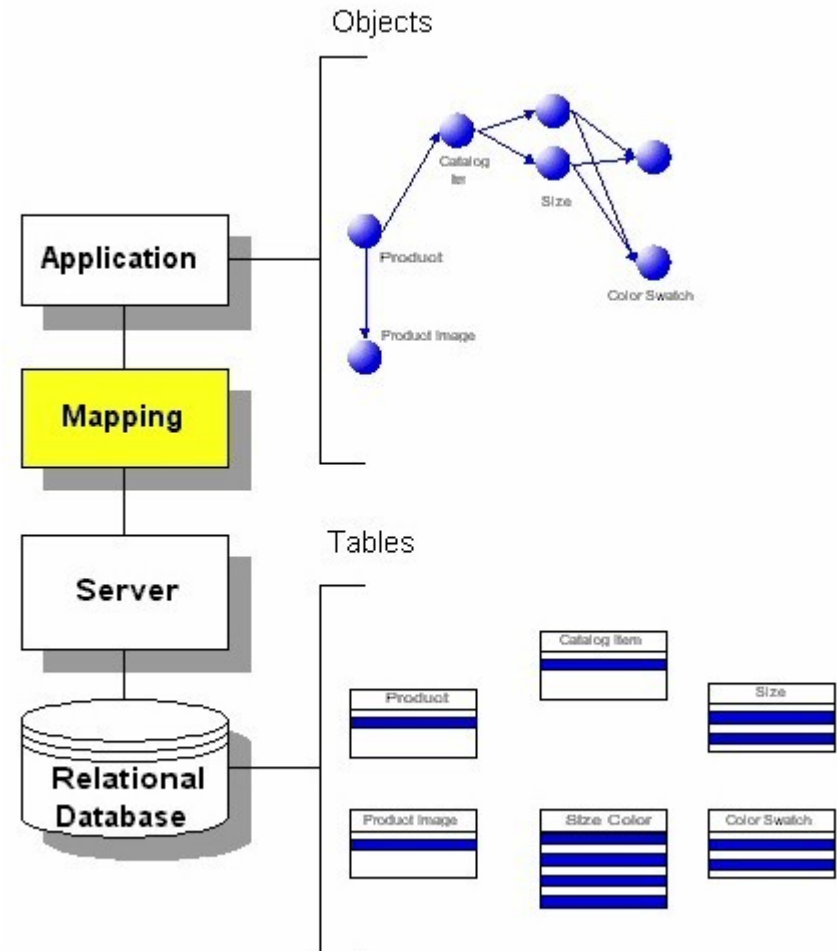


- JDBC (Java DataBase Connectivity)
- **Inconvénients :**
 - Nécessite l'écriture de nombreuses lignes de codes répétitives
 - La liaison entre les objets et les tables est un travail de bas niveau
- Exemple de code + pattern DAO

Impedance Mismatch

Désigne la difficulté de stocker des graphes d'objets interconnectés en mémoire sous forme de tables dans des système de base de données relationnelles (RDBMS)

- Granularité
- Héritage
- Egalité
- Association
- Navigation



Mapping relationnel-objet



Concept permettant de connecter un modèle objet à un modèle relationnel.

Couche qui va interagir entre l'application et la base de données.

Pourquoi utiliser ce concept?

- Pas besoin de connaître l'ensemble des tables et des champs de la base de données
- Faire abstraction de toute la partie SQL d'une application.

Mapping relationnel-objet (2)



Avantages :

- Gain de temps au niveau du développement d'une application.
- Abstraction de toute la partie SQL.
- La portabilité de l'application d'un point de vue SGBD.

Inconvénients :

- L'optimisation des frameworks/outils proposés
- La difficulté à maîtriser les frameworks/outils.

Critères de choix d'un ORM

- La facilité du mapping des tables avec les classes, des champs avec les attributs.
- Les fonctionnalités de bases des modèles relationnel et objet.
- Les performances et optimisations proposées : gestion du cache, chargement différé.
- Les fonctionnalités avancées : gestion des sessions, des transactions
- Intégration IDE : outils graphiques
- La maturité.

JPA

- Une API (Java Persistence API)
- Des implémentations



MyBatis



ORACLE
TOPLINK

JPA

- Permet de définir le mapping entre des objets Java et des tables en base de données
- Remplace les appels à la base de données via JDBC

Concepts vs Classes



<i>Concept</i>	<i>JDBC</i>	<i>Hibernate</i>	<i>JPA</i>
<i>Ressource</i>	Connection	Session	EntityManager
<i>Fabrique de ressources</i>	DataSource	SessionFactory	EntityManagerFactory
<i>Exception</i>	SQLException	HibernateException	PersistenceException

Configuration de Spring Hibernate



- Modification du contexte Spring pour la configuration :
 - dataSource (locale ou distante avec JNDI)
 - sessionFactory
 - transactionManager
 - hibernateTemplate

Supports de Spring



<i>Classes de support</i>	<i>JDBC</i>	<i>Hibernate</i>	<i>JPA</i>
<i>Classe template</i>	JdbcTemplate	HibernateTemplate	JPATemplate
<i>Classe de support de DAO</i>	JdbcDaoSupport	HibernateDaoSupport	JPADaoSupport
<i>Gestionnaire de transactions</i>	DataSourceTransactionManager	HibernateTransactionManager	JPATransactionManager

Objets Hibernate



- **SessionFactory** : Un cache threadsafe (immuable) des mappings vers une (et une seule) base de données. Une factory (fabrique) de Session et un client de ConnectionProvider. Peut contenir un cache optionnel de données (de second niveau) qui est réutilisable entre les différentes transactions que cela soit au niveau processus ou au niveau cluster.
- **Session** : Un objet mono-threadé, à durée de vie courte, qui représente une conversation entre l'application et l'entrepôt de persistance. Encapsule une connexion JDBC, une Factory (fabrique) des objets Transaction. Contient un cache (de premier niveau) des objets persistants, ce cache est obligatoire. Il est utilisé lors de la navigation dans le graphe d'objets ou lors de la récupération d'objets par leur identifiant.

Objets Hibernate (2)



- **Objets et Collections persistants** : Objets mono-threadés à vie courte contenant l'état de persistance et la fonction métier. Ceux-ci sont en général les objets métier ; la seule particularité est qu'ils sont associés avec une (et une seule) Session.
*Dès que la Session est fermée, ils seront détachés et libres d'être utilisés par n'importe quelle couche de l'application (i.e. de et vers la présentation en tant que Data Transfer Objects - DTO : objet de transfert de données).
- **Objets et collections transitoires (Transient)** : Instances de classes persistantes qui ne sont actuellement pas associées à une Session. Elles ont pu être instanciées par l'application et ne pas avoir (encore) été persistées ou elles n'ont pu être instanciées par une Session fermée.

Objets Hibernate (3)



- **Transaction** : (Optionnel) Un objet mono-threadé à vie courte utilisé par l'application pour définir une unité de travail atomique. Abstrait l'application des transactions sous-jacentes. Une Session peut fournir plusieurs Transactions dans certains cas.
- **TransactionFactory** : (Optionnel) Une fabrique d'instances de Transaction. Non exposé à l'application, mais peut être étendu/implémenté par le développeur.

HibernateTemplate



Gère toutes les interactions avec la base de données :

- Recherche : find, getReference
- Sauvegarde : persist, merge
- Suppression : remove
- Requêtage : langage JP-QL (HQL), createQuery,
- Transaction : begin, commit, rollback
(support des transactions par annotations)

Annotations JPA

- JPA est l'API de spécification de la couche de persistance au sein d'une application client lourd / léger.
(les spécifications EJB 3 ne traitent pas directement de la couche de persistance)
- JPA 2 est la partie de la spécification EJB 3.1 qui concerne la persistance des composants :

<http://download.oracle.com/otndocs/jcp/persistence-2.0-fr-oth-JSpec/>

Mapping relationnel



```
@Entity
@Table(name = "person")
public class Person implements Serializable {

    @Id()
    private long id;

    @Column(name = "first_name")
    private String firstName;

    @Temporal(TemporalType.DATE)
    @Column(name = "birth_day")
    private Date birthDay;

    public Person() {
        super();
    }

    // Getters / Setters
    ...
}
```

Objet géré en base

Nom de la table
mappant l'objet

Clé primaire

Nom de la colonne
mappant l'attribut

Attribut de type date

Gestion de la concurrence



La gestion de la concurrence est essentielle dans le cas de longues transactions. Hibernate possède plusieurs modèles de concurrence :
None – Optimistic (Versioned) – Pessimistic

<https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html>

- None : la transaction concurrentielle est déléguée au SGBD. Elle peut échouer.
- Optimistic (Versioned) : si nous détectons un changement dans l'entité, nous ne pouvons pas la mettre à jour.
@Version(Numeric, Timestamp, DB Timestamp) :
on utilise une colonne explicite Version (meilleure stratégie).
- Pessimistic : utilisation des LockMode spécifiques à chaque SGBD.

Gestion de la concurrence

Versioned



- L'élément @Version indique que la table contient des enregistrements versionnés. La propriété est incrémentée automatiquement par Hibernate.

Automatiquement, la requête générée inclura un test sur ce champ :

- ```
UPDATE Player SET version = @p0, PlayerName = @p1
WHERE PlayerId = @p2
AND version = @p3;
```

# Gestion de la concurrence

## Pessimistic



- Nous pouvons exécuter une commande séparée pour la base de données pour obtenir un verrou sur la ligne représentant l'entité :

```
player = session.get(Player.class,1);
session.Lock(player, LockMode.Upgrade);
player.PlayerName = "other";
tx.Commit();
```

- `SELECT PlayerId FROM Player with (updlock, rowlock) WHERE PlayerId = @p0;`
- `UPDATE Player SET PlayerName = @p1  
WHERE PlayerId = @p2 AND PlayerName = @p3;`
- Inconvénient : l'attente pour l'obtention du verrou (pour la modification si la ligne est verrouillée).  
Une exception est déclenchée après le Timeout parce que nous ne pouvons pas obtenir le verrou.

# Relations entre Entity Beans



- **One-To-One**  
@OneToOne  
@PrimaryKeyJoinColumn  
@JoinColumn
- **Many-To-One**  
@ManyToOne  
@JoinColumn
- **One-To-Many**  
@OneToMany  
(pas de @JoinColumn)
- **Many-To-Many**  
@ManyToMany  
@JoinTable

# Traitement en cascade

- Les annotations `@OneToOne`, `@OneToMany`, `@ManyToOne` et `@ManyToMany` possèdent l'attribut **cascade**
- Une opération appliquée à une entité est propagée aux relations de celle-ci :  
  
par exemple, lorsqu'un utilisateur est supprimé, son compte l'est également
- 4 Types : `PERSIST` , `MERGE` , `REMOVE` , `REFRESH`
- `CascadeType.ALL` : cumule les 4

# Héritage

Il existe trois façons d'organiser l'héritage :

- **SINGLE\_TABLE**  
@Inheritance  
@DiscriminatorColumn  
@DiscriminatorValue
- **TABLE\_PER\_CLASS**  
@Inheritance
- **JOINED**  
@Inheritance

La différence entre elles se situe au niveau de l'optimisation du stockage et des performances

# Héritage : SINGLE\_TABLE

- Tout est dans la même table
- Une seule table
- Une colonne, appelée “Discriminator” définit le type de la classe enregistrée.
- De nombreuses colonnes inutilisées

```
@Entity(name="COMPTE")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="compte_discriminator",
 discriminatorType=DiscriminatorType.STRING,length=15)
public abstract class Compte implements Serializable{...}
```

```
@Entity
@DiscriminatorValue("COMPTE_EPARGNE")
public class CompteEpargne extends Compte
 implements Serializable {...}
```

# Héritage : TABLE\_PER\_CLASS



- Chaque Entity Bean fils a sa propre table
- Lourd à gérer pour le polymorphisme

```
@Entity
@Inheritance (strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Compte implements Serializable
{...}
```

La clé @Id ne peut pas être @GeneratedValue (Identity)  
(le mettre en TABLE)

```
@Entity
public class CompteEpargne extends Compte
 implements Serializable {...}
```



# Héritage : JOINED

- Chaque Entity Bean a sa propre table
- Beaucoup de jointures

```
@Entity
@Inheritance (strategy=InheritanceType.JOINED)
public abstract class Compte implements Serializable
{...}
```

```
@Entity
public class CompteEpargne extends Compte
implements Serializable {...}
```

# Héritage : récapitulatif

| Stratégie            | SINGLE_TABLE                                | TABLE_PER_CLASS                | JOINED                                                                        |
|----------------------|---------------------------------------------|--------------------------------|-------------------------------------------------------------------------------|
| <b>Avantages</b>     | Aucune jointure,<br>donc très<br>performant | Performant en insertion        | Intégration des<br>données proche<br>du modèle objet                          |
| <b>Inconvénients</b> | Organisation des<br>données non<br>optimale | Polymorphisme lourd<br>à gérer | Utilisation<br>intensive des<br>jointures, donc<br>baisse des<br>performances |

# Requêtes



- JPA-QL (HQL)

<https://docs.jboss.org/hibernate/orm/3.3/reference/fr-FR/html/queryhql.html>

- Criteria

<https://docs.jboss.org/hibernate/orm/3.3/reference/fr-FR/html/querycriteria.html>

- SQL

<https://docs.jboss.org/hibernate/orm/3.6/reference/fr-FR/html/querysql.html>

- Requêtes nommées

# Gérer les exceptions



- **Récupérer les `DataAccessException`**

La cause de l'exception est l'erreur SQL.

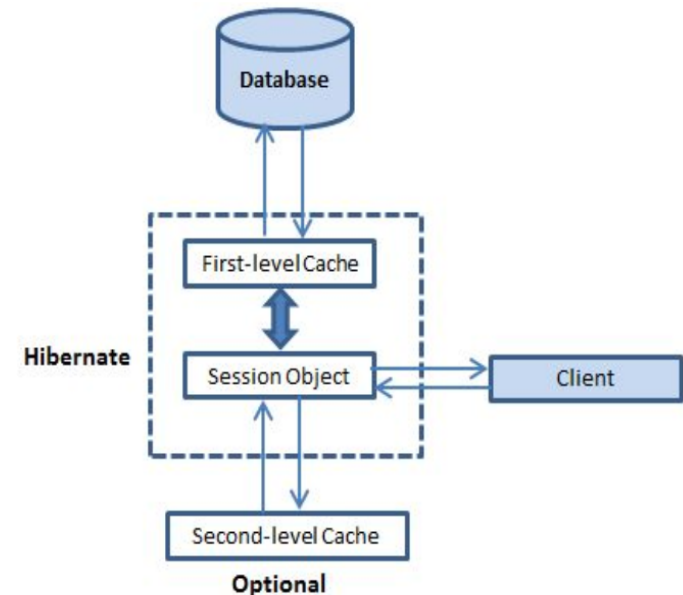
```
try {
 dao.delete(delete1);
} catch (DataAccessException e) {
 System.out.println(e.getMessage());
 System.out.println(e.getCause());
 SQLException sqllex = (SQLException) e.getCause();
 System.out.println(sqllex.getErrorCode());
 System.out.println(sqllex.getSQLState());
}
```

# Cache de niveau 1

NHibernate possède 2 niveaux de cache.

- **Cache de niveau 1 :**

- Chaque fois que vous passez un objet à `Save()`, `Update()` ou `SaveOrUpdate()` ou que vous récupérez un objet avec `Load()`, `Get()`, `List()`, ou `Enumerable()`, cet objet est ajouté au cache interne de la session (`ISession`).
- Quand `Flush()` est ensuite appelée, l'état de cet objet va être synchronisé avec la base de données. Si vous ne voulez pas que cette synchronisation se produise ou si vous traitez un grand nombre d'objets et la nécessité de gérer efficacement la mémoire :
  - La méthode **`evict()`** peut être utilisée pour supprimer l'objet et ses collections dépendantes du cache de premier niveau.
  - La méthode **`clear()`** permet de vider complètement le cache de la session.



# Cache de niveau 1 (suppression)



```
List<Book> books = session
 .createQuery("from Book").list();
 //un grand résultat

for (Book b : books)
{
 DoSomethingWithABook(b);
 hibernateTemplate.evict(b);
}
```

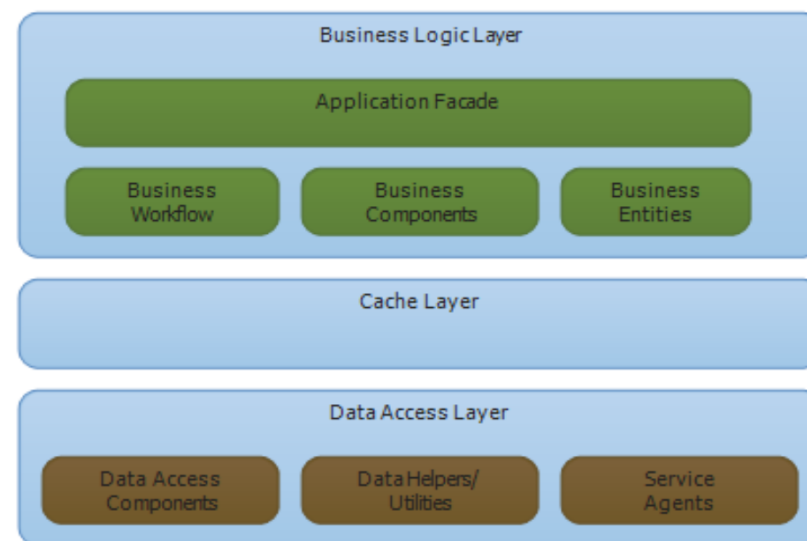
# Cache de niveau 2

## Introduction

- La mise en cache des entités est une technique très importante pour améliorer les performances de l'application. Généralement, on introduit une couche de mise en cache dans une architecture multi-couches **avant la couche d'accès aux données**.

Parfois, on utilise les composants web côté présentation pour mettre en cache les entités :

- Session
- Application (servletContext)



Hibernate fournit un cache de niveau 2 (couche accès aux données) qui permet de s'abstraire de l'utilisation des composants de la couche présentation ou métier.

# Cache de niveau 2 Configuration



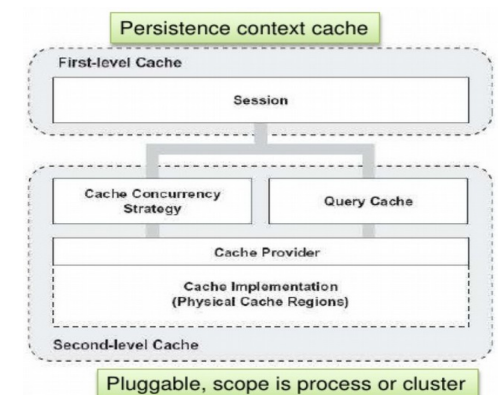
```
<property name="cache.use_second_level_cache">true</property>
<property name="cache.provider_class">...</property>
<property name="cache.use_query_cache">true</property>
<property name="prepare_sql">true</property>
```

- De multiples cache providers sont disponibles :

<https://docs.jboss.org/hibernate/stable/core.old/reference/fr/html/performance-cache.htm>

Exemple ehCache :

<http://www.baeldung.com/hibernate-second-level-cache>





# Cache de niveau 2

## Caching des entités



- Pour spécifier la mise en cache d'une entité ou d'une collection, on ajoute l'annotation `@Cache` dans le mapping de la classe ou de la collection ou une conf. Xml :
- L'attribut **usage** spécifie la stratégie de gestion de la concurrence :
  - **read only** : cache en lecture seule, pas de modification d'instances persistantes (manière la plus simple et la plus performante).
  - **read/write** : si l'application doit mettre à jour des données. On doit s'assurer que la transaction est terminée et que la session est fermée (strict).
  - **nonstrict read/write** : si l'application doit occasionnellement mettre à jour des données (multiples transactions simultanées).

# Cache de niveau 2

## Caching de requêtes



- Les résultats d'une requête peuvent être mis en cache.  
**Utile uniquement pour les requêtes exécutées fréquemment avec les mêmes paramètres.**
- Configuration :  
`cache.use_query_cache = true`
- Utilisation : `Query.setCacheable(true)`

