Spring Basics

Dependency Injection (par réflexion)

Setter : descriptive names, convention javabeans, héritables
Constructeur :dépendances obligatoires, immutabilité, concision (dans le programme java)

Construction des beans

Factory : `<bean factory-method=« maMethode»/>` `maMethod` doit être une méthode statique
FactoryBean : Object getObject() throw Exception, Class getObjectType(), boolean isSingleton()
Factory externe: `<bean factory-bean=« ... » factory-method=« ... »/>`. Doit être une méthode non statique

ApplicationContext:

Les définitions sont chargées depuis : classpath, FS, chemin dépendant de l'environnement

- ClasspathXmlApplicationContext
- FileSystemXmlApplicationContext
- XmlWebApplicationContent

Possibilité de forcer le mode de chargement avec les préfixes: classpath : , file : , http :
Bonne pratique de séparer les beans d'infrastructure des beans d'application

Résumé :

- Simplifie le code, améliore la testabilité
- Programmation par rapport à des interfaces
- Contrôle centralisé du cycle de vie des objets
- namespaces = couche d'abstraction des <bean> génériс dans un but de simplification (masque les détails) et expression plus parlante, validation semantic

Cycle de vie:

3 phases :

- **Initialization** : initialisation des beans, injection des dépendances...etc
- **Utilisation** : les beans sont utilisés par l'application
- **Destruction** : destructions des beans, destruction de l'ApplicationContext

Initialization:

Parsing des fichiers XML + lecture des définitions

`BeanFactoryPostProcessor.processBeanFactory(BeanFactory bf)` → modifie les définitions
Ex: PorpertyPlaceHolderConfigurer `<context:property-placeholder location=''...''/>`

Instanciation des beans (eager) → par réflexion

Paramétrage des beans (injection des dépendances) → par réflexion

BeanPostProcessor.postProcessBeforeInitialization(Object bean, String beanName)

Init : 3 façons de procéder

- @PostConstruct + `<context:annotation-config/>` (JSR 250) (recommandé)
- `<bean **init-method**=''maMethode''/>` (méthode publique sans arguments)
- Implémenter `InitializingBean` et redéfinir la méthode `afterPropertiesSet()`

```
BeanPostProcessor.postProcessAfterInitialization(Object bean, String
beanName)
```

BeanPostProcessors :

Détectés automatiquement par Spring

```
<context:annotation-config>
```

- `RequiredAnnotationBeanPostProcessor`
- `CommonAnnotationBeanPostProcessor`
- `AutowiredBeanPostProcessor`

Utilisation

Un BeanPostProcessor peut wrapper un bean Spring dans un proxy dynamic pour lui ajouter des comportements supplémentaires (transactions, sécurité, log....)

Destruction

3 façons de procéder

- @PreDestroy + `<context:annotation-config/>` (JSR 250) (recommandé)
- `<bean **destroy-method**=''maMethode''/>` (méthode publique sans arguments)
- Implémenter `DisposableBean` et redéfinir la méthode `destroy()`

Scopes :

- singleton (par défaut)
- prototype
- request (web)
- session (web)
- custom

Heritage de bean

```
<bean id=''pere'' class=''....'' abstract=''true''/>

<bean id=''fils'' class=''....'' parent=''pere''/>
```

La propriété ***abstract="true"*** indique à Spring de ne pas instancier ce bean

PropertyEditor

Prédéfinis : NumberEditor, BooleanEditor, DateEditor, RessourceEditor, PropertiesEditor....

Définir son propre Property Editor :

il faut étendre `PropertyEditorSupport` (java.beans.PropertyEditorSupport) et les méthodes

- String getAsText()
- void setAsText(String)

Enregistrement de l'éditor dans le fichier XML :

```
<bean class=''org....CustomEditorConfigurer''>
      <property name=''customEditors''>
            <map>
                  <entry key=''classe''><bean class=''editor''/></entry>
            </map>
      </property>
</bean>
```

NB: `CustomerEditorConfigurer` est un BeanFactoryPostProcessor

Namespace <util:>

<util:list>, <util:map>, <util:set> permettent de déclarer des collections standalone

Leurs attributs list-class, map-class et set-class permettent d'utiliser des collections génériques

<util:constant static-field="com.example.MyClass.HOST_NAME"/>

<util:properties location="classpath:mail.properties"/>

TEST

Pourquoi tester

Automatiser les tests → confiance → refactoring facile → agilité

Meilleur design et moindre coût de correction des bugs

Tests unitaires : test d'1 module en isolation. Utilisation de stubs . Test d'un scénario à la fois

Tests d'intégration : test inter-modules, avec l'architecture. Utilisation de mocks

Stubs vs Mocks: les stubs doivent implémenter toutes les méthodes de l'interface, pas les mocks (proxy)

Préférer les mocks pour les interfaces complexes ou pour vérifier les appels. Bonus pas de classe à écrire.

Junit

Junit 3.8 :

La classe doit : `extends AbstractDependencyInjectionSpringContextTests`
Redéfinir la méthode `String [] getConfigLocations()`

Junit 4 : les annotations

- @RunWith(SpringJUnit4ClassRunner.class)
- @ContextConfiguration(locations={"...", "... "})
- @Test, @Autowired, @Transactional

Annotations

Autowired

@Autowired([required=false])

@Qualifier("beanName")

@Resource("beanName") – JSR 250. Peut rechercher des objets JNDI

```
<context:annotation-config>
```

Composants

@Component(["name"])
ou @Repository, @Service, @Controller

```
<context:component-scan base-package=''...''/>
	<context:include-filter type=''regex|annotation|aspectj|assignable''
expression=''...''/>
	<context:exclude-filter type=''regex|annotation|aspectj|assignable''
expression=''...''/>
```

@Scope("prototype")

Spring AOP

Buts

- Modularisation des problématiques transverses (log, sécurité, cache....)
- Éviter le couplage fort (**code tangling**) et l'éparpillement (**code scattering**).

Concepts

- AspectJ : bytecode engineering, Spring AOP : proxies dynamique
- Joinpoint : événement du code
- Pointcut : expression qui sélectionne un ou plusieurs jointpoints
- Advice : code à exécuter à un jointpoint sélectionné par une expression poincut
- Aspect : module qui encapsule des pointcuts et des advices

Implémenter un Aspect

@Aspect, @Pointcut, @Around, @Before, @After, @AfterThrowing, @AfterReturning

```
<aop:aspectj-autoproxy>
      <aop:include name=''beanref vers l'aspect''/>
</aop:aspectj-autoproxy>
```

Contexte de joinPoint

La méthode d'aspect peut avoir comme premier argument un objet JointPoint

- this : le proxy
- target : l'objet ciblé par l'aspect
- args : les arguments passés à la méthode cible de l'aspect
- signature : la signature complète de la méthode cible de l'aspect

Syntaxe de pointcut

execution(*expr*) - *expr* [modifier] ReturnType [ClassType] MethodName([args]) [Throws ExceptionType]

Spring AOP utilise des proxies et ne s'applique donc qu'aux méthodes publiques

Arguments : **..** = 0..n - ***** = 1 de n'importe quel type - ClassType : .. = package*, + = extends

Bindings :

- this
- target
- args
- annotation

Around

La méthode d'advice annotée par @Around doit prendre un paramètre de type `ProceedingJoinPoint`

Appeler `proceed()` pour réellement invoquer la méthode cible.

Configuration XML sans les annotations

```
<aop:config>

	<aop:pointcut id=''setterMethod'' expression=''execution(void set*(*))''>

	<aop:aspect ref=''ref du bean d'aspect''>

		<aop:before pointcut-ref=''setterMethod'' method=''méthode d'aspect''/>

	</aop:aspect>

</aop:config>
```

Paramètres des annotations

@AfterThrowing (value="*pointcut*", throwing="**e**")
public void myAdvice(Exception **e**)

@AfterReturning (value="*pointcut*", returning="***result***")
public void myAdvice(Object **result**)

Spring Data Access

Features

- Élimination du code boilerplate → réduction des bugs, productivité améliorée
- Gestion des transactions de manière déclarative
- Gestion des ressources (connexions) → pas de fuite mémoire
- Gestion intelligente des exceptions
- Fonctionne de manière uniforme entre les technologies et les environnements
- Fournit toujours une Factory, un template et une classe Support

Couches de l'architecture

- **Service** : encapsule la logique métier
- **DAO** : masque la complexité de l'accès aux données, masque les détails d'implémentation
- **Infrastructure** : services de la plateforme technique. Varie selon les environnements (dev, prod...)

Fonctionnement

Une Factory crée la ressource (connexion...), et un TransactionManager la gère ensuite

Exemples:
- **JDBC**: BasicDataSource + DataSourceTransactionManager
- **Hibernate**: LocalSessionFactoryBean + SessionFactory + BasicDataSource + HibernateTransactionManager
- **JEE**: JNDIObjectFactoryBean + (managed) DataSource + JTATransactionManager

Spring JDBC

JDBCTemplate et SimpleJDBCTemplate

Prennent une Datasource en paramètre de constructeur
Sont Thread-safe → réutilisables dans plusieurs DAO
SimpleJDBCTemplate (java 1.5) est une version simplifiée, mais qui gère les types paramétrés
Possibilité de récupérer un JDBCTemplate "normal" avec getJDBCOperations()
Méthodes:
- `X queryForX(String sql, Object[] args)` avec X type primitif
- `Object queryForObject(String sql, Object[] args)`
- `Map queryForMap(String sql, Object[] args)` → Map<colonne, valeur>
- `List queryForList(String sql, Object[] args)` → liste de Map<colonne, valeur>
- `List query(String sql, Object[] args, [RowMapper | RowCallbackHandler | ResultSetExtractor])` → liste d'objets

Transformations:
- `RowMapper : Object mapRow(ResultSet rs, int index)`
- `RowCallbackHandler : void processRow(ResultSet rs)`
- `ResultSetExtractor : Object extractData(ResultSet rs)`

Classes Support: JdbcDaoSupport et SimpleJdbcDaoSupport
Méthodes : setDatasource / getJdbcTemplate ou getSimpleJdbcTemplate
Note : les classes Support ne fonctionnent pas bien avec @Autowired

Transactions

ACID

- **A**tomic : tout ou rien
- **C**onsistent : contraintes d'intégrité ok
- **I**solated : changements temporaires invisibles
- **D**urable : changements committés définitifs

Risques sans les transactions

- Partial failure :seule une partie des opérations est réalisée
- Dirty reads : voir les changements temporaires effectués dans d'autres transactions
- Lost updates : deux threads voient la même valeur initiale, le dernier qui écrit une modification gagne

Features

- Gestion déclarative des transactions par métadonnées (XML, annotations)
- Gestion uniforme quelque soit l'environnement et la technologie du DAO

Managers

L'interface principale est **`PlatformTransactionManager`**

Quelques exemples d'implémentations :

- `DataSourceTransactionManager`
- `HibernateTransactionManager`
- `JpaTransactionManager`
- `JtaTransactionManager`

Mise en place : 2 étapes

- Déclarer un TransactionManager (par ex: DataSourceTransactionManager)
- Déclarer les transactions, soit part **AOP** (`<tx:advice...>`), soit par **annotation** (@Transactional **+** `<tx:annotation-driven/>`**)**

Annotations : @Transactional(isolation=..., propagation=..., readonly=..., timeout=...)

XML : 2 étapes

Tout d'abord, définir un aspect transactionnel

```
<tx:advice id=''txAdvice>

	<tx:attributes>

		<tx:method name=''get*'' propagation=... isolation=... readonly=..../>

		<tx:method name=''*'' /> valeur par défaut pour toutes les méthodes non-get

	</tx:attributes>

</tx:advice>
```

Ensuite, appliquer cet aspect transactionnel à un pointcut

```
<aop:config>

      <aop:pointcut id=''myPointcut'' expression=''...''/>

      <aop:advisor pointcut-ref=''myPointcut'' advice-ref=''txAdvice''/>

</aop:config>
```

Transactions programmatiques

`TransactionTemplate` qui prend un TransactionManager en paramètre de constructeur

Exemple de code :

```
txTemplate.execute (new TransactionCallback ()
      {
            public Object doInTransaction(TransactionStatus status)
            {…}
      }
);
```

NB : voir aussi la définition de `TransactionStatus` dans l'API et ses méthodes proposées.

Propagation

- **REQUIRED (défaut)** : Utilise la transaction courante, ou en crée une nouvelle si besoin
- **REQUIRES_NEW** : Suspend la transaction courante si elle existe, et exécute une nouvelle transaction indépendante
- **MANDATORY** : Utilise la transaction courante si elle existe, si aucune transaction est présente, lève une exception `TransactionRequiredException`.
- **NEVER** : Si aucune transaction n'est présente, ne démarre aucune transaction, si une transaction existe, lève une `RemoteException`
- **SUPPORTS :** Utilise la transaction courante si elle existe, sinon ne démarre aucune transaction.

ORM et Hibernate

O/R concepts / problèmes

Granularité
- Modélisation fine en java, pour bien encapsuler les données métier
- Modélisation plus grossière (ou différente) en BDD pour les perfs

Identité
- Identité (= =) et équivalence (.equals) en java
- Clés primaires dans les BDD, indépendantes des données métier

Associations :
- limitées aux clés étrangères pour la BDD, alors que l'on a des relations uni et multi directionnelles en java + héritage

Bénéfices de l'ORM
- Query language
- Persistence automatique des graphes, détection du changement
- Mise en cache (L1 : par transaction, L2 : par SessionFactory)

Concepts Hibernate

- Session = unité de travail. Gère le cache de L1
- SessionFactory = représente la source de données. Thread-safe, réutilisable, gère le cache L2

Annotations

Gère JPA (java.persistence.*) et Hibernate (org.hibernate.*)

Utiliser les annotations standard (JPA) de préférence, étendre avec celles d'Hibernate

On peut annoter les classes, les méthodes (setters...) et les propriétés

Par défaut, tous les champs sont persistés, sauf si marqués @Transient

Annotations
- @Entity, @Table(name="*nomTable*") pour les classes
- @Id, @Column(name="*nomCol*") pour les propriétés

Par défaut le mode d'accès aux champs est "*field*" (accès direct aux champs)

XML

Configuration XML

```
<hibernate-mapping package=''x.y.z''>
      <class name=''classe'' table=''table''>
            <id name=''clientId'' access=''field''/>
            <property name=''firstName'' column=''FIRST_NAME''/>
      </class>
</hibernate>
```

Par défaut l'accès est en mode "*property*" (utilisation des setters)

Mapping des collections

@OneToMany (targetEntity = Client.class, cascade=CascadeType.ALL)

@JoinColumn (name="client_id")
private Set<Client> clients

```
<set name=''clients'' table=''CLIENTS'' cascade= ''all''>
     <key column=''client_id''/>
     <one-to-many class=''client''/>
</set>
```

Requêtage

```
Client c = (Client) session.get(Client.class, clientId);

Query q = session.createQuery(''From Client c where c.id =: id'') ;
q.setString(''id'', clientId);
List clients = q.list();
```

Mise en place d'Hibernate avec Spring

Pour configurer une SessionFactory, on a besoin :

- Une DataSource
- Les infos de mapping

Mapping par Annotation :

```
<bean class=''...AnnotationSessionFactoryBean''>
     <property name=''dataSource'' ref=''bean datasource''/>
     <property name=''annotatedClasses''>
          <list><value>x.y.Client</value></list>
     </property>

     OU

     <property name=''packagesToScan'' value=''x.y.Model''/>

</bean>
```

Mapping par XML :

```
<bean class=''...LocalSessionFactoryBean''>
     <property name=''dataSource'' ref=''bean datasource''/>
     <property name=''mappingLocations''>
          <list><value>classpath:example/Customer.hbm.xml</value></list>
     </property>
</bean>
```

Gestion des transactions

Il suffit d'utiliser les SessionFactoryBeans de Spring et de déclarer un TransactionManager (HibernateTransactionManager ou JtaTransactionManager) + `<tx:annotation-driven>`

Si on injecte la sessionFactory dans un DAO (au lieu d'un template), le DAO n'a aucune

dépendance Spring.

Gestion des exceptions

Les exceptions levées par les DAO "purs" sont celles d'Hibernate → AOP pour les transformer

En java 5+ :

@Repository + <bean class= "PersistenceExceptionTranslationInterceptor**PostProcessor**"/>

En java 4- : Utilisation de l'AOP

<bean id="***persistenceTranslator***" class="PersistenceExceptionTranslationInterceptor"/>

+

```
<aop:config>
        <aop:advisor advice-ref="persistenceTranslator" pointcut-ref="execution(...)"/>
</aop:config>
```

HibernateTemplate

- Gère les sessions (acquisition / libération)
- Transforme automatiquement les exceptions
- Fournit des méthodes utilitaires pour les tâches courantes : find, load, saveOrUpdate, delete
- Prend une SessionFactory comme argument de constructeur

HibernateDaoSupport

Définition de la classe : public class MyDao extends `HibernateDaoSupport`

Utiliser le template fournit par la classe Support : `getHibernateTemplate()`

Spring WEB

Philosophie

Doit être léger : ne sert qu'a effectuer le mapping HTTP request ↔ fonctionnalité métier

Doit être propre : Séparation entre logique (développeur) et apparence (designers)

Doit être flexible, facile à apprendre, testable

Autres modules

Spring Javascript
- Dégradation progressive
- Ajax
- Mise à jour par fragments, popups modales...

Spring Webflow
- C'est un contrôleur Spring MVC
- DSL pour définir les « flows »
- Chaque « flow » est une ressource accessible à une URL, et est isolé des autres utilisateurs

Spring JSF/Faces
- Utilise WebFlow pour la navigation
- Fournit une bibliothèque de composants
- Navigation simple, bonne gestion des exceptions, state management, Ajax

Chargement de l'ApplicationContext

Via un ContextListner + `<context-param>`

```
 <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEb-INF/app-config.xml</param-value>
</context-param>

 <listener>
      <listener-class>org...ContextLoaderListener</listener-class>
</listener>
```

Accès au contexte
- WebApplicationContextUtils.getRequiredWebApplicationContext(getServletContext())
- Étendre "ActionSupport" pour Struts + getWebApplicationContext()
- Dans les servlets Spring MVC, injection automatique avec @Autowired

Spring MVC

Front Controller : org....DispatcherServlet

Chaque servlet a son contexte propre (configuré avec l'init-param « contextConfigLocation »)

Les controlleurs sont annotés @Controller

Les méthodes sont annotés @RequestMapping("*/path/to/controller*")

Les méthodes handler renvoient un `ModelAndView` (méthodes: setViewName(), addObject(), …)
Si elles prennent des paramètres de type `HttpServletRequest/Response,` ils sont fournis automatiquement

Note MVC: le modèle représente le contrat entre le contrôleur et la vue

ViewResolver : par défaut, une `InternalRessourceView` avec prefix= "...", suffix="..."

Auto-détection des @Controller avec `<context:component-scan base-package=''...''/>`

Mapping automatique des paramètres de requête vers les paramètres des méthodes handler avec @RequestParam (ex: `public ModelAndView doIt(@RequestParam (''id'') long id)`)

Spring Remoting

Buts

- Cacher la complexité du code
- Exposition déclarative des services
- Utilisation uniforme quelque soit le protocole employé

Principes

Un exporter côté serveur

Un `ProxyFactoryBean` côté client. Les exceptions spécifiques sont traduites.

RMI

En RMI classique, le serveur expose un « skeleton » et le client compile un « stub » → invasif.

Avec Spring, plus besoin d'étendre les interfaces/classes « Remote »

Export

```
<bean class=''...RMIServiceExporter''>
      <property name=''serviceName'' value=''transferServiceRMI''>
      <property name=''serviceInterface'' value=''com...service.TransferService''>
      <property name=''service'' ref=''transferService''> (bean spring)
</bean>
```

Import

```
<bean id=''transferService'' class=''...RMIProxyFactoryBean''>
      <property name=''serviceInterface'' value=''com...service.TransferService''>
      <property name=''serviceUrl'' value=''rmi://localhost:1099/transferServiceRMI''>
</bean>
```

NB : Pour RMI/IIOP, on est toujours obligés de compiler les objets spécifiques(rmic)

HTTPInvoker / Burlap / Hessian

Export

```
<bean name=''/transferService'' class=''...(HTTInvoker|Burlap|Hessian)ServiceExporter''>
      <property name=''serviceInterface'' value=''com...service.TransferService ''>
      <property name=''service'' ref=''transferService''> (bean spring)
</bean>
```

Import

```
<bean id=''transferService'' class=''...(HTTInvoker|Burlap|Hessian)ProxyFactortBean''>
      <property name=''serviceInterface'' value=''com...service.TransferService''>
      <property name=''serviceUrl'' value=''monURL/transferService''>
</bean>
```

NB : Burlap et Hession a été crée par Caucho

Hessian uses binary XML
Burlap uses textual XML

Leur mécanisme de sérialisation/désérialisation Object/XML est basé sur un mécanisme propriétaire

Que choisir comme protocole

- Spring partout : HTTPInvoker (sérialisation java sur HTTP POST)
- Java sans Web : RMI
- Interopérabilité avec Web : Burlap (binaire) / Hessian (XML)
- Interopérabilité sans Web : Corba (RMI/IIOP)

Attention : Tous ces protocoles se basent sur du RPC, il faut donc que le client connaisse les détails du service exporté (méthodes, attributs, ..etc.). En java, il faut aussi que les versions des classes soient identiques

Si c'est problématique (ex : clients externes), utiliser les web services.

Accès au EJB (stateless SessionBeans)

Namespace

`<jee:local-slsb/>` ou`<jee:remote-slsb/>`

Exemple

```
<jee:remote-slsb id=''transferService''
                 jndi-name=''java:comp/env/ejb/transferService''
                 business-interface=''foo.ItransferService'' >

      <jee:environment>
            java.naming.provider.url = t3://remoteServer:7001
            java.naming.factory.initial = weblogic.jndi.WLInitialContextFactory
      </jee:environment>

</jee:remote-slsb>
```

Spring WEB SERVICES

Avantages

Couplage faible entre les systèmes → facilite le changement

Inter-opérabilité grâce au XML, traduction des exceptions

Contract-first : plus besoin de connaître les détails d'implémentation du service

Bonnes pratiques : validation tolérante, utilisation de XPath

Architecture

MessageDispatcher, Endpoints(mappés par des EndpointMappings), EndpointInterceptors

Il faut déclarer la servlet MessageDispatcher comme un front-controller

Spring Web Services ressemble beaucoup à Spring MVC

Spring Web MVC	**Spring Web Services**
DispatcherServlet	MessageDispatcher
HandlerMapping	EndpointMapping
Controller	Endpoint
HandlerInterceptor	EndpointInterceptor

Ecriture d'un Endpoint

Implémenter `PayloadEndpoint` : `public Source invoke (Source request)`
Configurer comme bean Spring

Mapping des Endpoints : 4 stratégies de mapping

Annotations (PayloadRootAnnotationMethodEndpointMapping)
QName de la racine (PayloadRootQNameEndpointMapping)
Xpath (XPathPayloadEndpointMapping)
Header HTTP « **SoapAction** » (SoapActionEndpointMapping)

Marshalling / Unmarshalling

Interfaces

- `Marshaller` : void marshal (Object graph, Result result) throws ...
- `Unmashaller` : void unmarshal (Source source) throws ...

Les implémentations se trouvent dans le package `org.springframework.oxm`

spring-oxm.jar peut être utilisé en standalone

les classes implémentent les deux interfaces `Marshaller` et `Unmarshaller`

Supporte JAXB1, JAXB2, Castor, JiBX, XmlBeans, XStream

Les marshaller/unmarshallers sont utilisés par :

- `AbstractMarshallingPayloadEndpoint` (côté serveur)
- `WebServiceTemplate` (côté client)

NB: fournir la même instance pour les propriétés `marshaller` et `unmarshaller` des beans WebserviceTemplate et AbstractMarshallingPayloadEndpoint

WebServiceTemplate

Travaille directement avec le XML

Utilise les marshallers / unmarshallers

Fournit des méthodes simples pour les opérations courantes et des callbacks pour le reste :

- `WebServiceMessageCallback : public void doWithMessage(WebServiceMessage message)`
- `WebServiceMessageExtractor : public Object extractData(WebServiceMessage message)`

Méthodes :

- `sendSourceAndReceiveToResult ([String uri], Source source, Result result)`
- `marshalSendAndReceive([uri], Object payload, [messageCallback]) → Object`
- `sendAndReceive([uri], messageCallback, messageExtractor)`

Spring Security

Concepts

- **Principale** : celui qui effectue l'action (homme ou machine) = identité
- **Authentification** : prouver que l'on est bien qui on doit être (basic, digest, X.509..etc)
- **Authorization** : contrôler les droits d'accès à la ressource. (basé sur des rôles)
- **Secured Item** : la ressource protégée

Buts

- **Portabilité** : avec ou sans conteneur ou serveur d'application
- **Flexibilité** : mécanismes d'authentification et d'autorisation configurables
- **Extensibilité** : définition des rôles, repository utilisé, détails de l'identité...
- **Séparation des rôles** : découplage du codes métier (AOP, filtres)

Fonctionnement

Le gestionnaire d'authentification remplit le contexte de sécurité (Principale + Credentials)

L'intercepteur de sécurité interroge des Voters qui décident de l'autorisation d'accès

Sécurité Web (Filtres)

Déclarer un filtre « springSecurityFilterChain » de type `DelegatingFilterProxy`

Activer les filtres

```
<security:http auto-config=''true''>
      <security:intercept-url pattern=''/*'' access=''ROLE_ADMIN''/>
      <security:form-login login-page=''/login.jsp''/>
</security:http>
```

Les patterns sont évalués dans l'ordre de déclaration : mettre les plus spécifiques en premier

Page de login :

le formulaire doit pointer sur "*j_spring_security_check*" et contenir les champs "*j_username*" et "*j_password*"

Taglibs :

```
<security:authentification property=''principale.username''>
<security:authorized if(Any|All|Not)Granted=''ROLE_ADMIN, ROLE_MEMBER''>
```

Authentification

DAO par défaut

```
<security:authentification-provider>
      <security:jdbc-user-service data-source-ref=''myDS''/>
      OU
      <security:user-service properties=''/WEB-INF/users.properties''/>
</security:authentification-provider>
```

Il est possible de configurer les requêtes SQL permettant de récupérer les users et roles

Sécurité des méthodes (AOP) : Par @Annotations ou en XML

@Annotations

- @Secured({"ROLE_MEMBER"}) et `<security:global-method-security secured-annotations=''enabled''/>`
- @RolesAllowed({"ROLE_MEMBER"}) et `<security:global-method-security jsr250-annotations=''enabled''/>`

XML :

```
<security:global-method-security>

      <security:protect-pointcut expression=''...'' access=''ROLE_MEMBER''/>

</security:global-method-security>
```

Ordre des filtres

- `HttpSessionContextIntegrationFilter`
- `LogoutFilter`
- `AuthentificationProcessingFilter`
- `ExceptionTranslationFilter`
- `FilterSecurityInterceptor`

Pour remplacer un filtre ou en insérer un dans la chaîne, il suffit d'ajouter à son <bean>
`<security : custom-filter (before|after|position)=''com..monFiltre''/>`

Spring JMS

Buts

- Portabilité : indépendance par rapport aux produits et éditeurs, grâce aux principe de message
- Flexibilité : utilisation d'un serveur JMS standalone ou de celui d'un serveur JEE

Concepts

- **Message** : Interface Message + TextMessage, ObjectMessage, MapMessage, BytesMessage, StreamMessage
- **Destination** : Queue (point à point) et Topic (publish / subscribe)
- **ConnectionFactory et Connection** : permettent d'obtenir une session sur le serveur JMS donné
- **Session** : unité de travail. Gère les transactions (session.commit(), session.rollback())
- **MessageProducer et MessageConsumer**

Exemples de configuration Spring

```
<bean id=''connectionFactory'' class=''ActiveMQConnectionFactory''>
      <property name=''brokerUrl'' value=''tcp://localhost:61616''/> ou vm://.... in-memory
</bean>

ou via JNDI : <jee:jndi-lookup id=''connectionFactory'' jndi-name=''jms/connectionFactory''/>


<bean id=''orderQueue'' class=''...ActiveMQQueue''>
      <constructor-arg value=''queue.order''/>
</bean>

ou via JNDI : <jee:jndi-lookup id=''orderQueue'' jndi-name=''jms/orderQueue''/>
```

Utilisation

Connection conn = connectionFactory.createConnection();
Session session = conn.createSession(boolean transacted, int acknowledgeMode);
Message msg = session.createXMessage(payload);
MessageProducer producer = session.createProducer(destination);
MessageConsumer consumer = session.createConsumer(destination);

ConnectionFactory (brokerUrl) → Connection → Session (Message OU (MessageProducer| MessageConsumer))

Envoi de message : JMSTemplate

Réduit le code initial, fournit des méthodes utilitaires pour les tâches courantes.

Traduit les exceptions

S'appuie sur :

- Un MessageConverter (SimpleMessageConverter)
- Un DestinationResolver (DynamicDestinationResolver) → Destination resolveDestinationNale(...)
- Une ConnectionFactory

Envoyer un message

- `convertAndSend(Object Message)`
- `convertAndSend(Destination dest, Object Message)`
- `convertAndSend(String destName, Object Message)`

Envoyer un message avec des callbacks

- `void send(MessageCreator creator)`
- `Object execute(ProducerCallback callback)`
- `Object execute(SessionCallback callback)`

MessageCreator : `Message createMessage(Session session)`
ProducerCallback : `Object doInJms(Session session, MessageProducer producer)`
SessionCallback : `Object doInJms(Session session)`

Réception des messages

Méthodes bloquantes de JmsTemplate

- `receive()`
- `Message receive ( [Destination dest | String destName ] )`
- `Object receiveAndConvert ( [Destination dest | String destName ] )`

Réception Asynchrone

L'API JMS propose l'interface `MessageListener` et redéfinir la méthode
`public void onMessage( Message m )`

NB : Traditionnellement l'utilisation d'un `MessageListener` requiert un conteneur EJB

Les listeners doivent implémenter `MessageListener` ou `SessionAwareMessageListener`

Pour configurer :

```
<jms:listener-container connection-factory=''ref to connection factory bean''>
      <jms:listener ref=''objet cible'' destination=''queue name'' />
</ jms:listener-container>
```

Spring message driven object

- `SimpleMessageListenerContainer`, utilise l'API JMS et crée un nombre fixe de Session
- `DefaultMessageListenerContainer`, ajoute les transactions

Pour configurer :

```
<jms:listener-container connection-factory=''ref to connection factory bean''>
```

```
      <jms:listener ref=''objet cible'' destination=''queue name'' method=''methode''
response-destination=''dest name''/>

</ jms:listener-container>
```

Spring JMX

Buts

Management (changement de la configuration au runtime)
Monitoring (supervision, alertes, …)

Architecture

MBeans : objets managés (attributs + opérations)
MBean Server : contient les définitions des Mbeans (Map : ObjectName → Mbean)
Connecteur JSR-160 : permettent de communiquer avec le serveur de Mbeans (RMI,etc...)

Utilisation

Déclarer un MbeanServer. Il est possible d'en utiliser un déjà existant (Server JEE...)

```
<bean id=''mBeanServer'' class=''...MBeanServerFactoryBean''>
      <property name=''localeExistingServerIfPossible'' value=''true''/>
</bean
```

Exporter les beans à l'aide d'un MbeanExporter : facilite l'enregistrement des beans

MbeanExporter utilise un **ObjectNamingStrategy** pour nommer les Mbeans (défaut : `KeyNamingStrategy`)

MbeanExporter utilise un **MBeanInfoAssembler** pour déterminer quoi exporter (défaut : `SimpleReflectiveMBeanInfoAssembler`)

```
<bean class=''...MBeanExporter''>
      <property name=''beans''>
            <map>
                  <entry key=''...'' value-ref=''..''/>
            </map>
      </property>
      <property name=''namingStrategy'' ref=''strategy''/>
      <property name=''assembler'' ref=''assembler''/>
</bean>
```

Naming Strategy (3)

- `KeyNamingStrategy (défaut)` : utilise la clé de la Map comme nom de bean
- `IdentityNamingStrategy` : utilise l'identité de l'objet dans la JVM
- `MetadataNamingStrategy` : utilise un @ManagedRessource(objectName= « foo »)

MbeanInfoAssembler (4) : se base sur la spécification des « Model MBeans »

- `SimpleReflectiveMBeanInfoAssembler` : expose les propriétés et méthodes publiques par réflexion
- `MethodNameBasedMBeanInfoAssembler` : possède une propriété « managedMethods » qui indique quelles méthodes exposer
- `InterfaceBasedMBeanInfoAssembler` : possède une propriété « managedInterfaces » qui indique quoi exposer. A Noter : le bean n'est pas obligé d'implémenter ces interfaces !
- `MetadataMBenInfoAssembler` : utilise @ManagedOperation et @ManagedAttribute

Export d'autres bean

Certains beans d'infrastructures sont déjà des Mbean
- ActiveMQ ConnectionFactory
- Hibernate StatisticsService

Utiliser `<context:mbean-export/>` pour rechercher et exporter les Mbeans existants.