

Spring Framework

Mise en œuvre

Support de cours

Réf. T462-025





Module 0

A propos de cette formation

Votre formateur

- Son nom
- Ses activités
- Ses domaines de compétence
- Ses qualifications et expériences pour ce cours

Votre formation - Présentation

- Description
 - Framework incontournable de la galaxie Java / Java EE, Spring à su s'imposer grâce à sa faculté à simplifier l'approche de programmation Java EE. En remettant en cause les lourdeurs de la plateforme Java EE, il propose aujourd'hui des alternatives pour toutes les couches d'une architecture logicielle n-tiers.
 - Découvrons ensemble les principales briques fonctionnelles de ce framework.
- Profil des stagiaires
 - Développeurs Java / Java EE, Chefs de projets, Architectes Java / Java EE
- Connaissances préalables
 - Expérience pratique du développement Java et Java EE
 - Connaissance de la plateforme Java EE
- Objectifs à atteindre
 - Comprendre le positionnement de Spring dans les technologies Java.
 - Développer une application en respectant le découpage en couches applicatives et le modèle MVC.
 - Comprendre le principe de fonctionnement et la configuration du conteneur Spring.
 - Utiliser les différents modules de Spring Framework.

Votre formation - Programme

- Introduction à Spring
 - Historique de Spring Framework / Fonctionnalités / Versions
 - Positionnement dans les technologies Java / Son conteneur léger / Spring vs. EJB
 - Bonnes pratiques d'architecture et de conception - La réponse apportée par Spring Framework
 - Les modules et projets Spring
 - Vers une simplification de l'approche Java EE
- Spring Framework : Le conteneur léger
 - La notion de conteneur léger
 - Le cycle de vie et le contexte Spring
 - La portée et le cycle de vie des objets (bean)
 - loc et injection de dépendances
- Configuration des applications Spring
 - Principes de configuration d'une application Spring / Java SE / Java EE
 - La configuration par fichiers XML / *-context.xml
 - La configuration par annotations
 - La définition des beans Spring / Dépendances / Héritage / Propriétés simples / Collections / Externalisation
- L'accès aux données dans Spring Framework
 - Rappels fondamentaux sur les techniques d'accès aux données en Java / JDBC / ORM
 - La prise en charge de JDBC dans Spring et ses avantages / Spring JDBC / DAO / JdbcTemplate / Annotations / Exceptions
 - Le mapping Objet/Relationnel avec Spring / JPA / Hibernate

Votre formation - Programme

- Les tests avec Spring
 - Tests avec Spring
 - Configuration des tests
 - Spécificités dans les tests
- Les applications Web - Spring MVC
 - Présentation de Spring MVC
 - Configuration d'une application Web Spring MVC
 - Les contrôleurs
 - Les vues
- Services Web - Spring Rest
 - Présentation de Spring Rest
 - Configuration
 - Les contrôleurs Rest
 - Les formats de données
 - Récupération des données
- Gestion de la réponse
- Gestion des exceptions
- Intégration avec JMS
 - Les technologies d'intégration
 - La messagerie applicative JMS
- La sécurité des applications Spring
 - Rappels sur la sécurité des applications Java avec JAAS
 - La sécurité dans Spring avec Spring Security (filtres de sécurité/URLs/annotations)
 - Configuration de l'authentification et des autorisations d'accès
 - La sécurité appliquée à l'invocation des beans
 - Implémenter Spring Security dans les JSP

Votre formation - Programme

- Les tests avec Spring
- Les applications Web – Spring MVC
- Spring Rest
- Intégration avec Spring
- La sécurité des applications Spring

Tour de table – Présentez-vous

- Votre nom
- Votre société
- Votre métier
- Vos compétences dans des domaines en rapport avec cette formation
- Les objectifs et vos attentes vis-à-vis de cette formation



Module 1

Introduction à Spring

Contenu du module

- Qu'est-ce que Spring ?
- Historique de Spring Framework
 - Evolutions des fonctionnalités
 - Les différentes versions
- Le positionnement de Spring dans les technologies Java
 - Les solutions apportées par Spring grâce à son conteneur léger
 - Comparaison avec la technologie EJB
- Les bonnes pratiques d'architecture et de conception d'application et la réponse apportée par Spring Framework
- Les différents modules et projets Spring
- Vers une simplification de l'approche Java EE

Qu'est ce que Spring ?

- Initialement, Spring est un conteneur dit « léger »
 - Une infrastructure similaire à un serveur d'application Java EE qui prend donc en charge
 - la création d'objets, leurs cycles de vie, leurs suppression ;
 - la mise en relation de ces objets ;
 - Le tout par l'intermédiaire d'une configuration
 - L'avantage notable par rapport aux serveurs d'application est que les classes n'ont pas besoin d'implémenter des interfaces, ou d'hériter de classes pour être prises en charge.
 - Les classes sont de simples POJO (Plain Old Java Object) ou Beans (En référence au JavaBeans)
- Aujourd'hui Spring est un projet à part entière...
 - Le framework (qui contient le conteneur) ;
 - Des projets et modules additionnels pour couvrir les divers besoins d'un développement d'applications d'entreprise.

<http://spring.io>

Historique de Spring Framework

- Depuis ses débuts en 1998, la plateforme Java Enterprise est très largement adoptée dans les développements d'applications d'entreprises complexes.
- Mais le foisonnement de ces développements sans véritable modèle architectural fédérateur est à l'origine des impressions de lourdeur et de complexité de cette plateforme.
- Prise de conscience et réflexions.....
 - Formalisation de principes et de modèles architecturaux
 - Développement par composants ;
 - Concepts de conteneurs légers (par opposition aux conteneurs Java EE, notamment EJB, jugés trop lourds) ;
 - Inversion de contrôle ;
 - Réduction du couplage et injection de dépendances.
- Ce sont les points de départ des travaux de Rod Johnson en 2002 et 2003
 - Il sort, en 2003, un ouvrage aujourd'hui Best Seller, « Expert One-on-One J2EE »
 - Les débuts du framework Spring

Problématiques

- Java EE est une infrastructure complexe
 - Un bon niveau technique est nécessaire pour maîtriser la plateforme
 - API riches
 - Composants (EJB) complexes
 - La technique prend souvent le dessus sur le fonctionnel
 - Conséquence : Manque de productivité dans les développements
 - Des couches d'abstraction sont régulièrement redéveloppées pour faciliter le travail des développeurs
- Java EE n'encourage pas la séparation des préoccupations
 - Problématiques techniques
 - Problématiques métiers
- Java EE impose une plateforme d'exécution lourde
 - Problème d'interopérabilité entre les différentes implémentations
 - Côté EJB principalement
 - Inadéquation des conteneurs par rapports aux besoins de l'application
 - Manque de modularité
- Java EE ne facilite pas les tests des développements
 - Forte dépendance à la (lourde) plateforme
 - Multiplication des techniques et frameworks de tests (jamais pleinement satisfaisants)
 - Car les tests sont forcément fait « In-container »

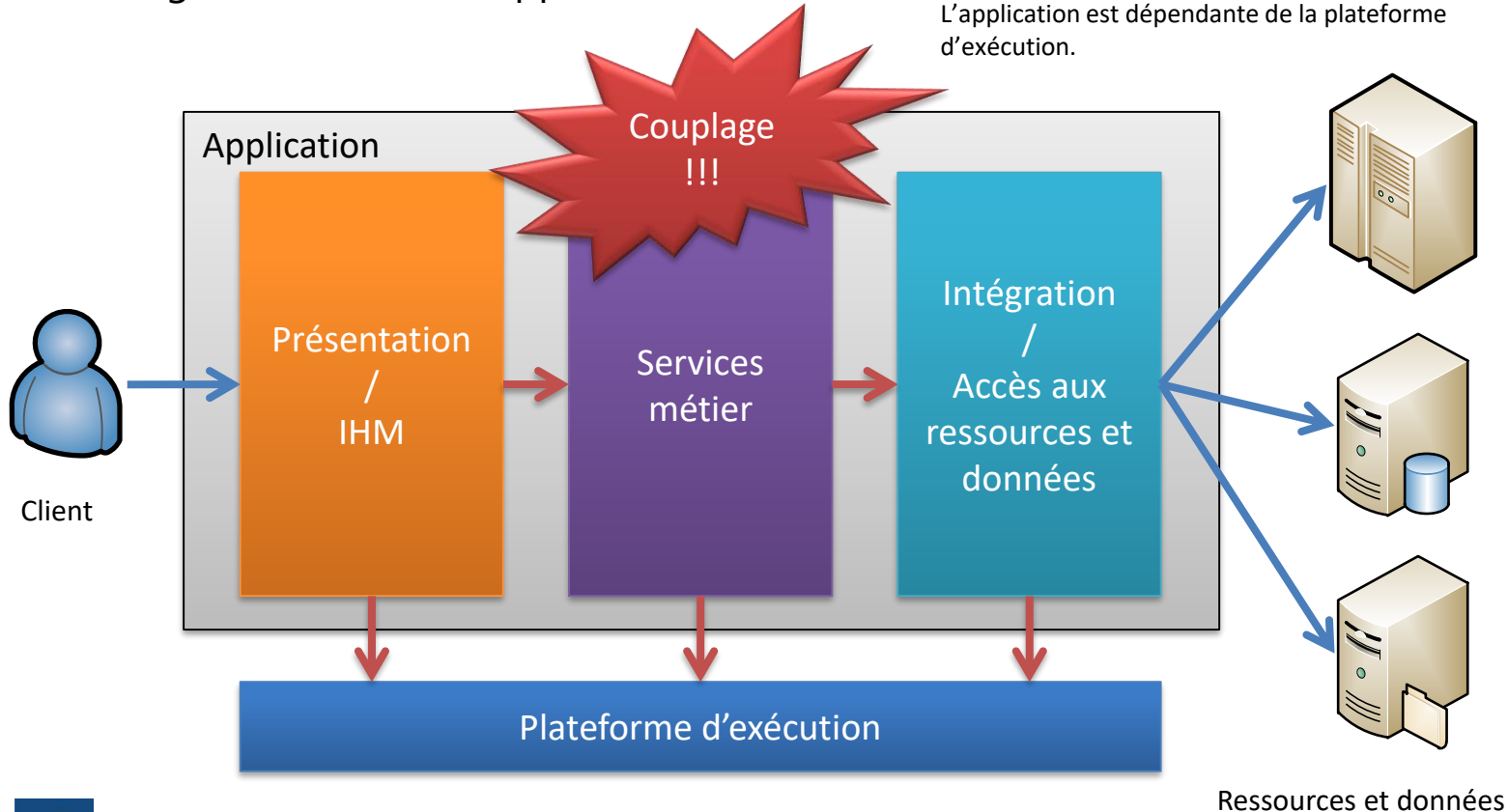
Solutions apportées par Spring Framework

- Le conteneur « léger »
 - Par opposition aux conteneurs Java EE, notamment les conteneurs EJB
 - Lourds
 - Intrusifs dans le code (couplage)
 - La technologie EJB est dépendante du serveur d'application (configuration)
 - Un conteneur léger est suffisant dans de nombreux projets
 - Ne dépend pas de Java EE (Utilisable dans les projets Java SE)
 - Le couplage est géré par le conteneur, plus par les composants (Facilite les tests)
 - Utilisation d'interface (Spécification/Implémentation)
 - Le conteneur léger injecte les dépendances entre les différents composants de l'application
- La programmation orientée aspects (AOP)
 - Permet d'offrir des services similaires à ceux des conteneurs EJB (sans leur lourdeur)
 - Applicables à tout types d'objets Java !
 - Modulariser les éléments logiciels transversaux
 - Ils sont souvent dupliqués, car il est difficile de les centraliser avec une simple approche objet
 - L'aspect apporte une nouvelle approche de modularisation en étendant les concepts objets
 - Séparation des préoccupations
 - Orienté configuration
- L'intégration de framework tiers
 - 3 niveaux d'abstraction possibles
 - Intégration du framework dans le conteneur. Le conteneur fournit la configuration au framework
 - Disponibilité d'un « modèle » d'utilisation du framework. On simplifie les appels à l'API du framework
 - Abstraction de l'API native du framework. Pour normaliser l'utilisation de framework répondant à des besoins similaires
 - Cas typique de l'accès aux données

Architecture et conception d'applications

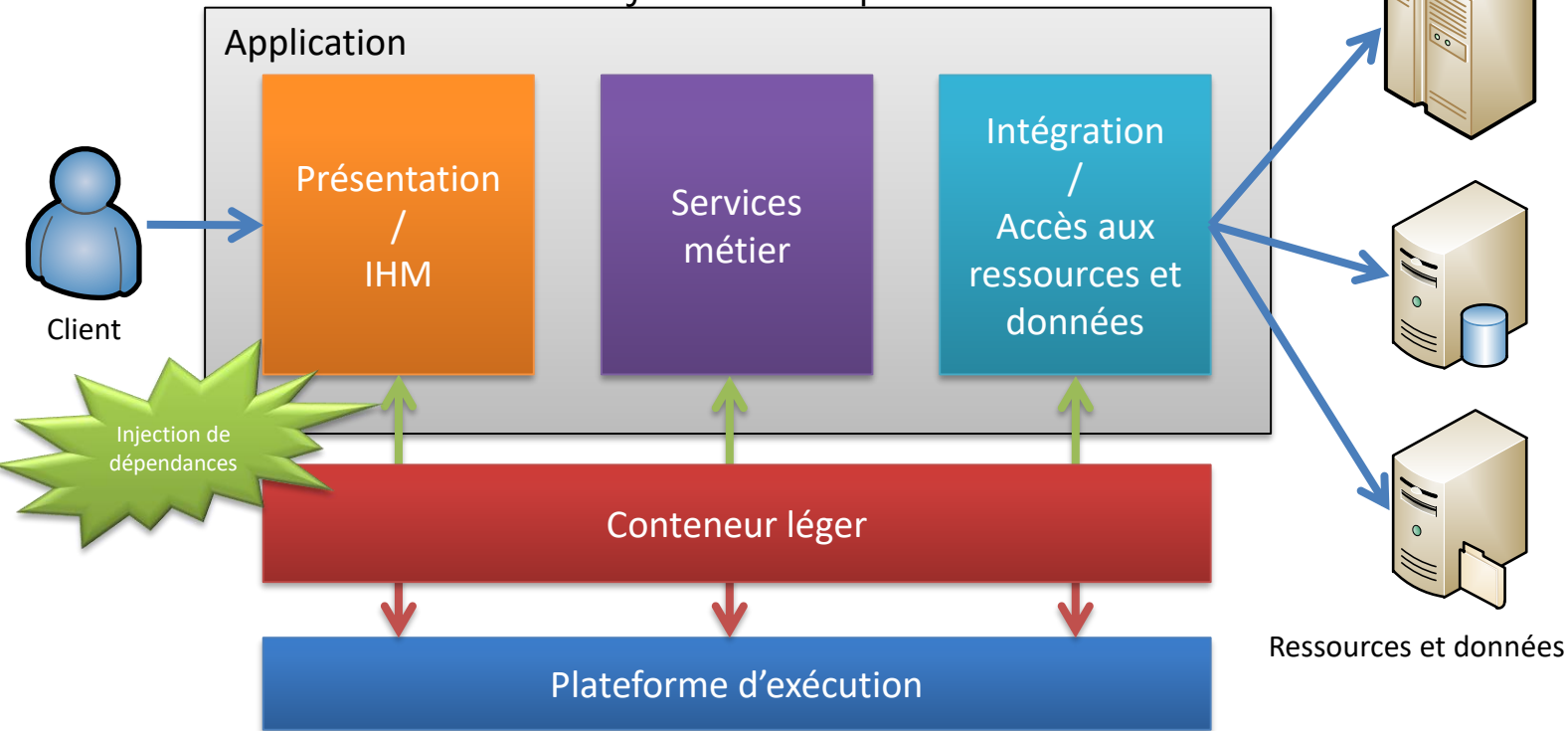
- Vue générale du développement en couche

Les couches applicatives sont interdépendantes.
L'application est dépendante de la plateforme d'exécution.



Architecture et conception d'applications

- Introduction d'un conteneur léger
 - Inversion de contrôle et injection de dépendances

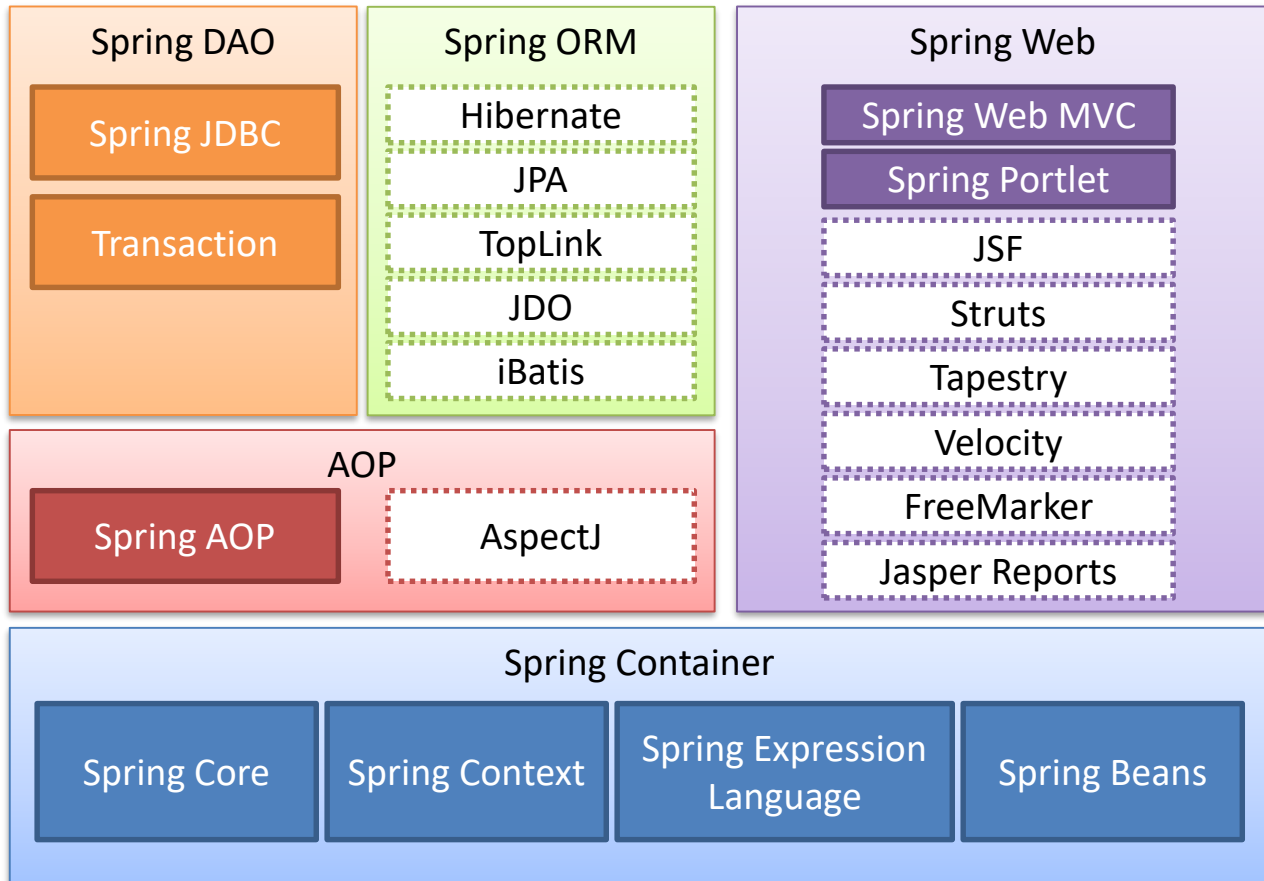


Les couches applicatives ne sont plus fortement couplées.
Le conteneur léger absorbe la dépendance à la plateforme d'exécution.

Les différents modules et projets Spring

- Le projet « fondation » : Spring Framework
 - Spring Container
 - Spring Core
 - Spring DAO / Spring ORM
 - Spring AOP
 - Spring Web / Spring Web MVC
- Les modules d'architecture applicative
 - Spring Web Flow
 - Construction d'application Web RIA/AJAX
 - Spring Security
- Les modules d'intégration applicative
 - Spring Web Services
 - Spring Batch
 - Mise en place de traitements par lots
 - Spring Dynamic Modules
 - Prise en charge des infrastructures OSGi
 - Spring Integration
 - Module pour l'EAI, une implémentation d'ESB
- Les modules de développement
 - Spring ROO
 - Composants de haut niveau et génération d'applications
 - Spring IDE
 - Plugin Eclipse pour le développement Spring
 - Spring STS
 - Environnement de développement complet pour Spring
 - Spring Bean Doc
 - Génération de la documentation d'un projet Spring
- Les modules d'ouverture technologique
 - Spring Rich Client
 - Module RCP basé sur Swing
 - Spring .Net
 - Intégration de Spring dans les applications Microsoft .NET
 - Spring BlazeDS
 - Intégration Adobe Flex

Architecture de Spring Framework



Vers une simplification de l'approche Java EE

- L'introduction d'un nouveau framework dans un développement d'application est souvent perçu comme un point de complexité supplémentaire.
- Spring Framework permet au contraire d'absorber cette complexité en rendant (plus ou moins) transparente l'utilisation des différents framework nécessaires au développement (Web, ORM, ...)
 - Il rend, de ce fait, également transparent les changements intervenants dans l'architecture (changement de framework par exemple).
- Cette abstraction de la complexité technique permet de se concentrer sur des préoccupations métiers et d'éviter de développer par soi-même ces couches d'abstraction techniques.
- L'utilisation d'un simple conteneur Web (tel Tomcat) pour héberger la solution Java EE est alors suffisante.
 - Le conteneur léger vient alors remplacer les services d'un conteneur d'EJB
- La couverture technique et la modularité de Spring permettent, aujourd'hui, de l'utiliser pour toutes les couches applicatives, uniformisant ainsi l'approche de développement d'un système complexe.



Module 2

Spring Framework : Le conteneur léger

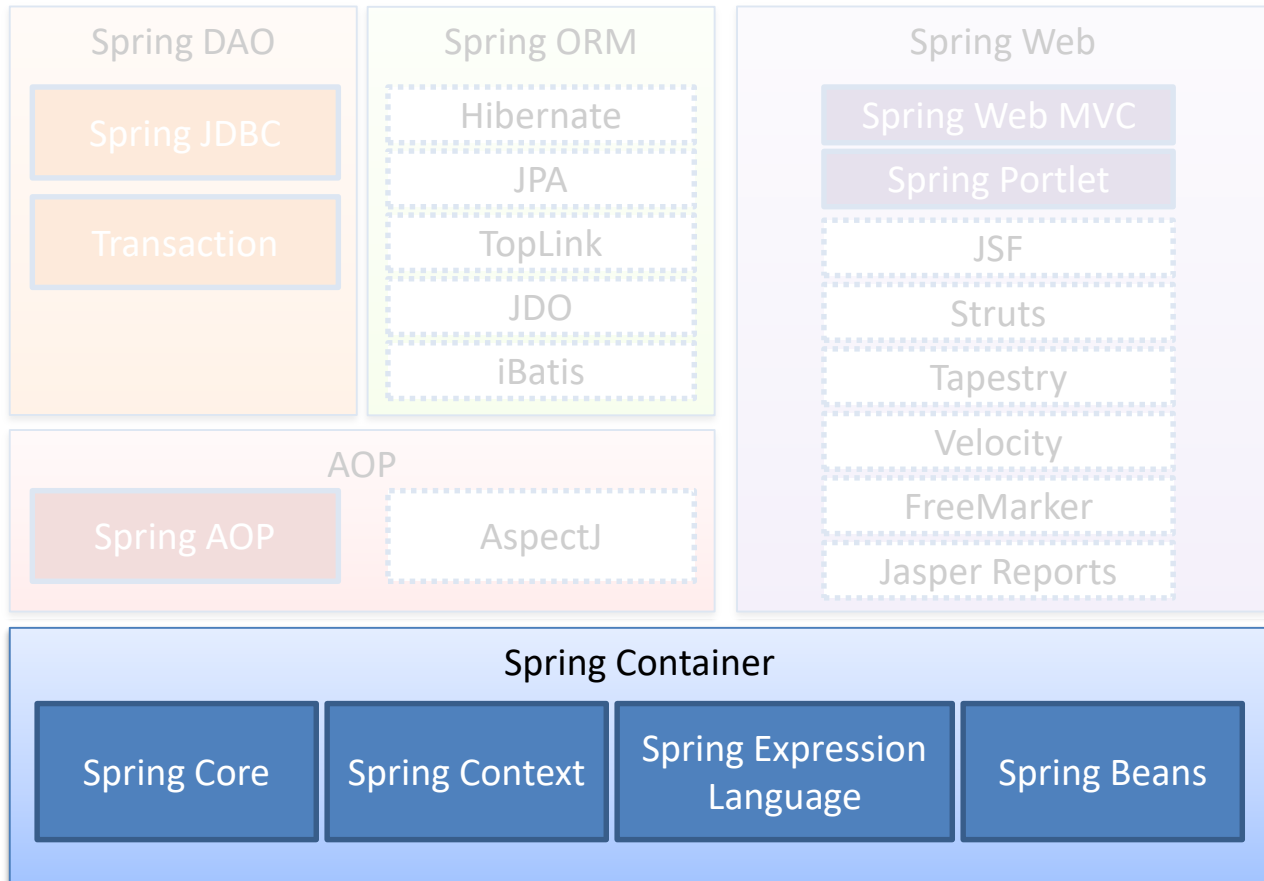
Contenu du module

- La notion de conteneur léger
 - Différence par rapport aux conteneurs lourds
- Le cycle de vie et le contexte Spring
 - Les différentes phases du cycle
 - Les interactions possibles
 - Les fabriques (BeanFactory) et contexte d'application (ApplicationContext)
- La portée et le cycle de vie des objets (bean)
 - Les « scopes » Singleton et Prototype par les patterns
- Ioc et injection de dépendances
 - Principes
 - Techniques
 - Dans Spring ...

Spring Core

- Spring Core constitue le socle applicatif de base du framework Spring
- Il est constitué du conteneur léger et des bibliothèques permettant de mettre en œuvre :
 - La journalisation
 - La fabrication des objets
 - L'injection de dépendance
- Spring Context est un autre composant indispensable au fonctionnement de base du framework. Le plus souvent, on ajoute Spring Context en tant que dépendance à son projet et Spring Core est inclus par dépendance.

Architecture de Spring Framework



La notion de conteneur léger

- Un conteneur léger c'est quoi ?
 - Un conteneur applicatif, c'est-à-dire une structure logicielle prenant en charge le cycle de vie (création, destruction, ...) des composants logiciels d'une application. Il est également responsable de la mise en œuvre de la collaboration entre ces composants.
- Pourquoi léger ?
 - Par opposition aux conteneurs Java EE jugés lourds
 - Des fonctionnalités bien souvent trop riches par rapport à ce qui est effectivement nécessaire
 - Impact sur la structure des composants (ils doivent implémenter des interfaces spécifiques)
 - Les services, à priori standard, le sont plus ou moins
- Le cas du conteneur Spring
 - Les composants pris en charge sont de simples classes Java.
 - Les services offerts aux composants sont fournis sous forme de modules d'extensions facultatifs au fonctionnement du conteneur.
 - Réutilise des mécanismes existant plutôt que d'obliger à les redévelopper (intégration de frameworks et librairies tiers).
 - Recommande l'utilisation d'interfaces pour le développement des composants. La séparation de la spécification (interface) et de l'implémentation (classe) permet de se focaliser sur le comportement du composant plutôt que sur sa manière de réaliser les actions correspondantes.

Cycle de vie et contexte Spring

- Une des responsabilités fondamentale du conteneur Spring est de gérer le cycle de vie des objets
 - 2 aspects :
 - La gestion des Singletons
 - La génération d'événements
 - Gestion des Singletons
 - Le Design-Pattern « Singleton » est implémenté sur les classes ne devant avoir qu'une seule instance au sein de l'application
 - Le conteneur léger de Spring peut, par simple configuration, considérer n'importe quelle classe Java comme devant être gérée en Singleton, sans impact sur le code.
 - Génération d'événements
 - Souvent un cycle de vie simple pour les objets : Création / Destruction
 - Mais possibilité d'invoquer des méthodes en fonction des changement d'état de l'objet
 - Après l'instanciation
 - Avant la destruction
 - Fonctionnalité importante quand la création et la destruction sont prise en charge à l'extérieur du code de l'application...

Les fabriques (BeanFactory)

- Le conteneur Spring doit donc fournir une structure permettant de récupérer les objets qu'il crée
 - C'est le rôle d'une fabrique de composant.
 - Implémentation du Design-Pattern « Factory »
 - Dans Spring Framework, les composants sont appelés « Beans » (en référence aux JavaBeans)
 - La fabrique est une interface spécifique du framework : BeanFactory
 - Elle définit les fonctionnalités dont l'application à besoin pour interagir avec le conteneur.
 - Plusieurs implémentations sont disponibles.
 - Cette interface est associée à un registre de définition des composants.
 - L'interface BeanDefinitionRegistry.
 - Elle fournit les fonctionnalités d'enregistrement des Beans dans le conteneur.
- Implémentations :
 - BeanFactory
 - DefaultListableBeanFactory : Implémentation de base.
 - XmlBeanFactory : Définition des objets dans des fichiers XML

Le contexte d'application (ApplicationContext)

- Le contexte d'application Spring permet d'étendre les fonctionnalités de fabrique et de registre de composants :
 - Externalisation et internationalisation des messages.
 - Chargement des ressources de l'application (fichiers...).
 - Publication / Notification d'événements applicatifs.
 - De plus, il peut y avoir plusieurs contextes hiérarchisés (pour séparer les couches notamment)
- ApplicationContext travaille avec une configuration.
 - Fichier(s) XML : en général nommés `applicationContext.xml`, `*-context.xml`,...
- Implémentations (associées à XmlBeanFactory)
 - FileSystemXmlApplicationContext
 - ClassPathXmlApplicationContext
 - XmlWebApplicationContext (Implémentation spécifique pour les applications Web Java EE)

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("applicationContext.xml");  
UnBean unBean = (UnBean) context.getBean("monBean");
```


Les composants – « Beans »

- D'un point de vue du conteneur, les composants applicatifs sont appelés Beans
 - En référence aux JavaBeans (ou encore POJO : Plain Old Java Object).
- Etant responsable de leur cycle de vie, il doit en connaître la définition.
 - Par configuration !
- Il est ensuite récupérable via le contexte d'application et utilisant son identifiant unique :
 - `userManager userManager = (userManager) context.getBean("userManager");`
- Par défaut avec Spring, les beans créés sont des singletons, plusieurs appels à `getBean()` renvoient donc la même instance.

Déclaration des beans

- En XML :

```
<beans>  
  <bean id="userManager" class="fr.formation.spring.service.UserManagerImpl" />  
  ...  
</beans>
```

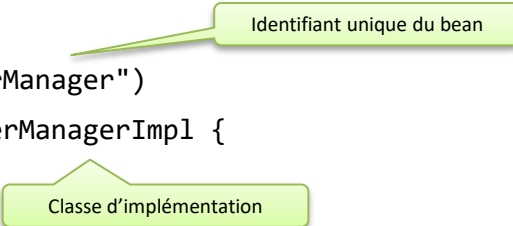


Identifiant unique du bean

Classe d'implémentation

- Avec les annotations :

```
package fr.formation.spring.service;  
  
import org.springframework.stereotype.Component;  
  
@Component("userManager")  
public class UserManagerImpl {  
  ...  
}
```



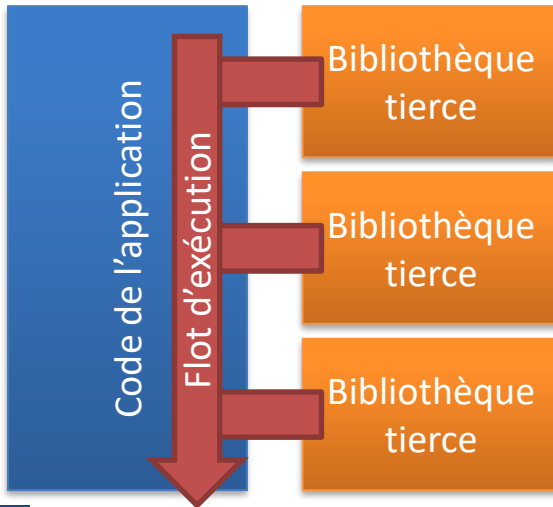
Identifiant unique du bean

Classe d'implémentation

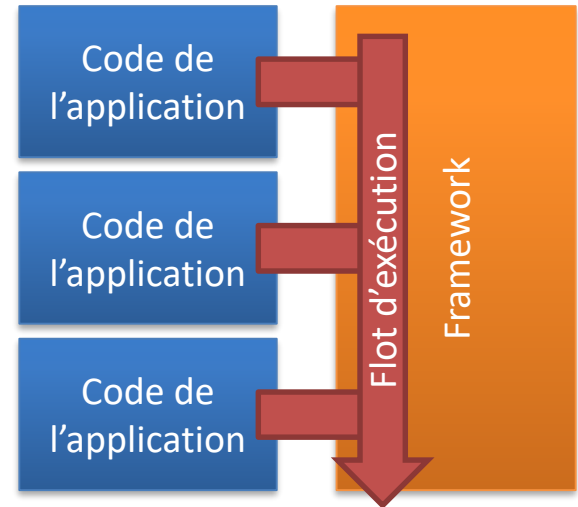
L'inversion de contrôle

- L'inversion de contrôle est le concept du conteneur Spring qui apporte et permet l'injection de dépendances.
 - Il s'agit d'une mise en œuvre du Design-Pattern « IoC » (Inversion of Control)
 - La notion de contrôle fait ici référence au flot d'exécution de l'application.
- C'est le conteneur qui pilote le flot d'exécution !

Sans inversion de contrôle



Avec inversion de contrôle



L'inversion de contrôle est aussi appelée principe « Hollywood » :
« Ne nous appelez pas, nous vous appellerons ! »

Dans Spring Framework...

- A partir du moment où le conteneur est instancié, il va prendre en charge la construction des objets au moment où ils sont nécessaires dans l'application, et les injecter dans les objets qui requièrent ces dépendances.
- Le faible couplage apporté par le framework (grâce à la séparation de la spécification et de l'implémentation) fait que les différents composants logiciels (et à fortiori, les différentes couches applicatives) ne communiquent plus directement.
 - Les appels passent forcément par le framework !
- En ce sens, le flot d'exécution est contrôlé par Spring
- Cependant, l'inversion de contrôle et l'injection de dépendances dans Spring ne signifie pas que le framework contrôle tout !
 - Dans certains cas, l'injection de dépendances ne fonctionne pas, il faudra avoir alors recours à une approche classique (appelée « recherche de dépendances »).
 - Spring Framework n'oblige pas à utiliser ces concepts, mais il les préconise et les facilite grandement.

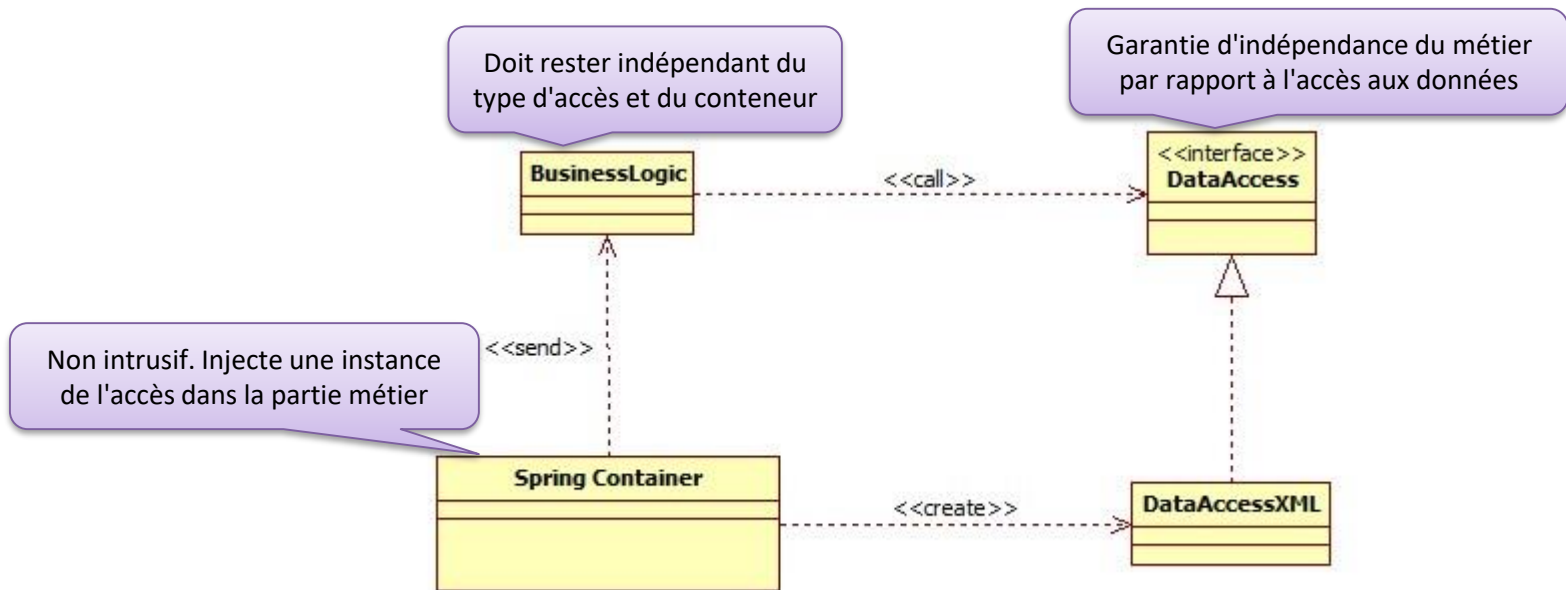
Différentes méthodes d'injection de dépendances

- Les méthodes les plus utilisées :
 - par mutateur (méthode setXXX)
 - par constructeur
- Par mutateur
 - Les dépendances entre les instances sont plus faibles
 - davantage de flexibilité
 - Représentatif de la relation d'agrégation
 - Association facultative
 - ré-injection possible
- Par constructeur
 - Ne permet pas la ré-injection
 - Association obligatoire
 - Moins flexible
 - Représentatif de la relation de composition
- Certains framework permettent également l'injection par interface, mais cette méthode reste marginale.

L'injection de dépendances

- Au-delà de gérer le cycle de vie des objets, le conteneur Spring se charge aussi d'établir les dépendances entre eux.
 - Ceci afin d'éviter un couplage trop important entre les objets, notamment entre leurs classes d'implémentation.
 - Ils seront donc essentiellement référencer dans l'application via leurs interfaces.
- L'injection de dépendance est donc un rôle fondamental du conteneur Spring.
- Elle est réalisée par le conteneur en fonction de la configuration des beans.
 - L'objet `ApplicationContext` réalise les injections à partir de la configuration Spring
 - Les beans sont liés entre-eux uniquement au niveau de leurs interfaces
 - L'association interface/classe est faite par le conteneur au moment de l'injection

Intérêt de l'injection de dépendances



L'implémentation n'est pas explicitement référencée dans le code, elle peut donc changer facilement !

Mise en œuvre de l'injection de dépendances - XML

- Code Java

```
public class BusinessLogic {  
  
    private DataAccess theDataAccess;  
  
    public void setTheDataAccess(DataAccess da) {  
        theDataAccess = da;  
    }  
}
```

- Configuration Spring

```
<beans>  
    <bean id="dataAccess" class="a.b.c.data.DataAccessXML" />  
    <bean id="businessLogic" class="a.b.c.BusinessLogic">  
        <property name="theDataAccess" ref="dataAccess" />  
    </bean>  
</beans>
```

Mise en œuvre de l'injection de dépendances - Java

- Déclaration du bean à injecter :

```
@Component("dataAccess")
public class DataAccessXML {

    ...

}
```

- Injection

```
public class BusinessLogic {

    @Autowired
    @Qualifier("dataAccess") // Si plusieurs implémentation de DataAccess
    private DataAccess theDataAccess;

    // Accesseur inutile avec l'injection Java !

}
```

Travaux Pratiques



www.eni-service.fr



Module 3

Configuration des applications Spring

Contenu du module

- Le contexte d'application
 - Spring dans les applications Java
 - Spring dans les applications Web
- La Configuration
- Externalisation des propriétés de configuration

Configuration des applications Spring

- La mise en œuvre de la configuration des applications Spring dépend :
 - Du format de configuration
 - Fichiers XML
 - Code Java + Annotations
 - Du type d'application
 - Application Java
 - Application Web Java EE

Spring dans les applications Java

- La classe principale sera responsable de démarrer le conteneur en instanciant la classe de contexte appropriée.
- Avec les fichiers XML, le contexte d'application est matérialisé par la classe `ClassPathXmlApplicationContext` :

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("applicationContext.xml");
```

- Avec la configuration Java, le contexte d'application est matérialisé par la classe `AnnotationConfigApplicationContext` :

```
ApplicationContext context =  
    new AnnotationConfigApplicationContext(MaConfig.class);
```

Spring dans les applications Web

- Spring est mis en œuvre de façon événementielle au chargement du contexte Java EE via un listener dans le fichier web.xml

```
<listener>  
  <listener-class>  
    org.springframework.web.context.ContextLoaderListener  
  </listener-class>  
</listener>
```

- Ce *listener* est notifié du démarrage de l'application Web par le serveur d'application et démarre le conteneur Spring
- On précisera ensuite la configuration avec des paramètres de contexte de Servlet

Spring dans les applications Web - Configuration

- Avec une configuration XML :

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/applicationContext-persistence.xml,
    /WEB-INF/applicationContext-metier.xml
  </param-value>
</context-param>
```

- Avec une configuration Java :

```
<context-param>
  <param-name>contextClass</param-name>
  <param-value>
    org.springframework.web.context.support.AnnotationConfigWebApplicationContext
  </param-value>
</context-param>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    fr.formation.banque.config.ConfigurationPersistence,
    fr.formation.banque.config.ConfigurationMetier
  </param-value>
</context-param>
```

La configuration

- La configuration des applications Spring consiste essentiellement à fournir des données au framework afin qu'il prennent en charge des créations d'objets complexes
- Ces objets seront ensuite utilisés par l'application
- Ces objets de configuration sont des beans déclarés comme des composants applicatifs
- Ces déclarations de beans sont constituées :
 - D'un ID unique et d'une classe d'implémentation comme pour un composant d'application
 - De propriétés qui servent à valoriser des attributs de la classe d'implémentation (via les setters)

Déclaration XML

- Exemple de déclaration XML - JPA
 - Une source de données et une EntityManagerFactory

```
<!-- Définition du bean de source de données pour Spring ORM (JPA) -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost:3306/banque?serverTimezone=Europe/Paris" />
  <property name="username" value="banque-user" />
  <property name="password" value="Pa$$w0rd" />
  <property name="initialSize" value="5" />
  <property name="maxTotal" value="20" />
</bean>

<!-- Définition du bean de EntityManagerFactory -->
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="jpaVendorAdapter">
    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="showSql" value="true" />
      <property name="databasePlatform" value="org.hibernate.dialect.MySQL8DBDialect" />
      <property name="packagesToScan" value="fr.formation.banque.persistance.entity" />
    </bean>
  </property>
</bean>
```

Déclaration Java

- Les classes de configuration sont annotées avec `@Configuration`
- Les méthodes déclarant des beans sont annotées avec `@Bean`
- La localisation des composants applicatifs est automatique avec l'annotation `@ComponentScan` indiquant les packages à analyser

```
@Configuration
@ComponentScan({
    "fr.formation.banque"
})
public class ConfigurationPersistence {

    @Bean("dataSource")
    public DataSource dataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/banque?serverTimezone=Europe/Paris");
        dataSource.setUsername("banque-user");
        dataSource.setPassword("Pa$$w0rd");
        dataSource.setInitialSize(5);
        dataSource.setMaxTotal(20);
        return dataSource;
    }

    @Bean("entityManagerFactory")
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

        LocalContainerEntityManagerFactoryBean entityManagerFactoryBean =
            new LocalContainerEntityManagerFactoryBean();

        entityManagerFactoryBean.setDataSource(dataSource());
        entityManagerFactoryBean.setPackagesToScan("fr.formation.banque.persistence.entity");

        HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        entityManagerFactoryBean.setJpaVendorAdapter(vendorAdapter);

        Properties props = new Properties();
        props.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL8DBDialect");
        props.setProperty("hibernate.show_sql", "true");

        entityManagerFactoryBean.setJpaProperties(props);

        return entityManagerFactoryBean;
    }
}
```

Externalisation des propriétés

- Il est possible grâce au « PropertyPlaceholderConfigurer » d'externaliser des propriétés de configuration dans des fichiers de type « properties ».
- Il faut déclarer cet objet et lui associer un (ou plusieurs) fichier(s) .properties
 - Les fichiers .properties doivent se trouver dans le CLASSPATH
- Le contenu des fichiers *properties* peut être automatiquement injecté dans un objet de type `Environment` grâce aux annotations `@PropertySource` et `@Autowired`
- Il est ensuite possible d'utiliser les propriétés grâce à la méthode `getProperty()` de l'objet `Environment`

@PropertySource

jpa.properties X

```
db.driver=com.mysql.cj.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/banque?serverTimezone=Europe/Paris
db.username=banque-user
db.password=Pa$$w0rd
db.pool.init=5
db.pool.max=20
```

```
@Configuration
@ComponentScan({
    "fr.formation.banque.persistance"
})
@PropertySource({
    "classpath:/jpa.properties"
})
public class ConfigurationPersistance {

    @Autowired
    private Environment env;

    @Bean("dataSource")
    public DataSource dataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName(env.getProperty("db.driver"));
        dataSource.setUrl(env.getProperty("db.url"));
        dataSource.setUsername(env.getProperty("db.username"));
        dataSource.setPassword(env.getProperty("db.password"));
        dataSource.setInitialSize(Integer.parseInt(env.getProperty("db.pool.init")));
        dataSource.setMaxTotal(Integer.parseInt(env.getProperty("db.pool.max")));
        return dataSource;
    }
}
```


Travaux Pratiques



www.eni-service.fr



Module 4

L'accès aux données dans Spring Framework

Contenu du module

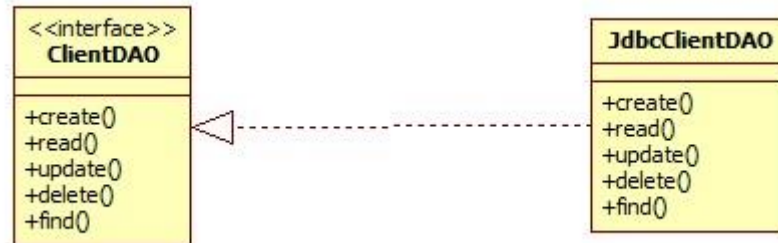
- Accès aux données en Java
 - Rappels et modèles d'implémentation
- Spring JDBC
 - Intérêt de Spring JDBC
 - La déclaration d'une DataSource
 - Utilisation de JdbcTemplate
- Spring ORM
 - Intérêt de Spring ORM
 - La configuration
 - DataSource et EntityManagerFactory
 - Spring Data JPA
- La gestion des transactions

Rappels fondamentaux sur l'accès aux données en Java

- L'accès aux bases de données se fait, en Java, grâce à l'API JDBC.
 - JDBC est une API de bas niveau permettant un accès universel aux bases de données.
 - Une application JDBC nécessite forcément l'utilisation d'un pilote d'accès à la base de données cible.
- L'inconvénient majeur de JDBC est la proportion de code à écrire pour arriver à ses fins.
 - Cependant, cela permet une maîtrise totale du code d'accès aux données.
- Un certain nombre de moyens sont aujourd'hui mis en œuvre pour faciliter l'approche d'accès aux données en Java, et notamment, le mapping Objet/Relationnel (ORM).
 - Le mapping objet/relationnel permet d'associer des entités objets (Java) à des ressources (tables) du modèle relationnel. La manipulation de données se faisant exclusivement au travers des entités.
- Implémentations ORM :
 - Hibernate : Framework Open Source (Aujourd'hui géré par RedHat/Jboss)
 - JPA (Java Persistence API) : Standard Java, initialement inspiré d'Hibernate
 - Mybatis : Framework Open Source de la fondation Apache

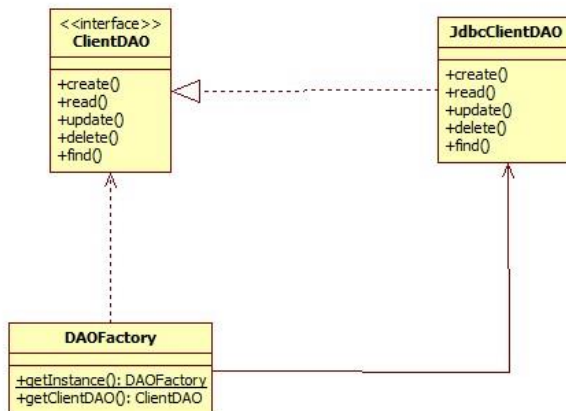
Conception de couches d'accès aux données

- Lors de la conception d'une couche d'accès aux données, il est obligatoire d'utiliser une API spécifique en fonction du type d'entrepôt dans lequel sont stockées les données.
 - SGBRD, XML, ...
- Cependant, les autres couches applicatives n'ont pas besoin de connaître ces spécificités
 - Pour réduire la complexité ;
 - Pour réduire le couplage, c'est-à-dire l'interdépendance entre les couches.
- Il est donc fondamental de n'exposer aux autres couches logicielles, que les aspects fonctionnels de l'accès aux données.
 - Il faut donc découpler la vue des clients sur les données (avec une interface) de l'implémentation spécifique (dans une classe).
 - C'est le principe de base du modèle de conception « Data Access Object » (DAO)



Data Access Object

- D'un point de vue du client (code appelant les méthodes d'accès aux données), les fonctionnalités ne sont disponibles qu'à travers de l'interface.
 - Problème : Une interface est abstraite et ne peut être instanciée !
 - Solution : Instancier la classe d'implémentation et renvoyer une représentation de l'objet créé sous-forme de l'interface.
 - `ClientDAO dao = new JdbcClientDAO();`
- Ceci est problématique car l'on induit une dépendance à la classe d'implémentation dans le code appelant
 - Solution : Travailler avec une classe intermédiaire servant de fabrique d'objet DAO.
- Avec Spring Framework, la fabrique est fournie !



Spring JDBC

- Le développement d'une application JDBC passe assez souvent par l'implémentation de méthodes dans lesquelles vont se trouver un grand nombre d'instruction relativement redondantes.
- Le schéma classique de programmation est le suivant :
 - Définir les paramètres de connexion
 - Ouvrir la connexion
 - *Ecrire la requête SQL*
 - Préparer et exécuter la requête
 - Mettre en place une boucle afin d'itérer sur les résultats (s'il y en a)
 - *Faire le traitement pour chaque itération*
 - Traiter les exceptions
 - Gérer les transactions
 - Fermer la connexion
- Le framework Spring peut prendre en charge tous les aspects bas niveau de ce modèle de programmation et permettre ainsi au développeur de se concentrer sur les aspects fonctionnels (en *italique*).

La connexion à la base de données

- Avec Spring JDBC, il est possible de définir un bean chargé de configurer la connexion à la base de données et de fournir une source de données (DataSource) à l'application.
 - Les propriétés de ce bean sont définies dans la configuration Spring
 - La source de données est ensuite injectée dans le bean métier.
- Une implémentation possible est la bibliothèque DBCP du projet Apache Jakarta Commons.
 - DBCP permet d'offrir une source de données associée à un pool de connexion aux applications
- Dépendances :
 - `spring-jdbc`
 - Apport `spring-tx`
 - Le pilote JDBC de la base de données

Un bean de source de données - DataSource

- On pourra externaliser les informations de connexion dans un fichier *properties* chargé dans la configuration avec `@PropertySource`
- Un objet `Environment` annoté avec `@Autowired` permettra d'accéder aux propriétés

```
@Autowired
private Environment env;

@Bean("dataSource")
public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName(env.getProperty("db.driver"));
    dataSource.setUrl(env.getProperty("db.url"));
    dataSource.setUsername(env.getProperty("db.username"));
    dataSource.setPassword(env.getProperty("db.password"));
    dataSource.setInitialSize(Integer.parseInt(env.getProperty("db.pool.init")));
    dataSource.setMaxTotal(Integer.parseInt(env.getProperty("db.pool.max")));
    return dataSource;
}
```

Spring JDBC

- Bien que Spring JDBC offre à l'application la possibilité de s'abstraire de la gestion de la connexion, le reste du code JDBC est à prendre en charge par le développeur.
 - Créer un objet `PreparedStatement` à partir de la connexion
 - Lier les paramètres à l'objet `PreparedStatement`
 - Exécuter l'objet `PreparedStatement`
 - Traiter les résultats
 - Gérer les `SQLException`
- Pour permettre de simplifier ces opérations, Spring Framework la classe `JdbcTemplate`
 - Permet de disposer de méthodes prenant en charge les opérations JDBC et la gestion des exceptions
 - L'objet est créé dans la configuration puis injecté dans les classes de manipulation de données

```
@Bean
public JdbcTemplate jdbcTemplate() {
    return new JdbcTemplate(dataSource());
}
```

Spring JDBC - JdbcTemplate

- La classe JdbcTemplate propose un ensemble de méthodes pour l'accès aux données aussi bien en lecture qu'en modification.
- Exemple de modification :

```
public class JdbcClientDAO implements ClientDAO {  
  
    @Autowired  
    private JdbcTemplate jdbcTemplate;  
  
    @Override  
    public void ajouterClient(Client client) {  
        String sql = "INSERT INTO Client(nom,prenom,adresse,codepostal,ville,motdepasse) VALUES(?,?,?,?,?,?)";  
        jdbcTemplate.update(sql,  
            new Object[] {  
                client.getNom(),  
                client.getPrenom(),  
                client.getAdresse(),  
                client.getCodePostal(),  
                client.getVille()  
            }  
        );  
    }  
}
```

Spring JDBC – JdbcTemplate pour interroger

- La récupération de données avec le JdbcTemplate passe par l'utilisation des méthodes query()
- Il existe plusieurs méthodes query() surchargées permettant de récupérer des résultats sous diverses formes.
 - Récupérer un résultat unique
 - Méthode queryForObject()
 - Récupérer plusieurs lignes
 - Méthode query() renvoyant une liste d'objet
- Ces méthodes utilisent un objet RowMapper pour la correspondance enregistrement/objet

Spring JDBC – JdbcTemplate – RowMapper

- Un RowMapper est un objet issue d'une classe implémentant l'interface `org.springframework.jdbc.core.RowMapper` et redéfinissant la méthode `mapRow()`
- Le corps de cette méthode contient le code permettant d'associer les données d'un enregistrement avec un objet Java.

```
public class ClientRowMapper implements RowMapper<Client> {  
  
    @Override  
    public Client mapRow(ResultSet resultSet, int rowNum) throws SQLException {  
        Client client = new Client();  
        client.setId(resultSet.getInt("id"));  
        client.setNom(resultSet.getString("nom"));  
        client.setPrenom(resultSet.getString("prenom"));  
        client.setAdresse(resultSet.getString("adresse"));  
        client.setCodePostal(resultSet.getString("codepostal"));  
        client.setVille(resultSet.getString("ville"));  
        client.setMotDePasse(resultSet.getString("motdepasse"));  
        return client;  
    }  
}
```

Spring JDBC – JdbcTemplate – Un seul résultat

- La méthode `queryForObject()` est une variante de la méthode `query()` permettant l'association directe de l'enregistrement avec l'objet, grâce au `RowMapper`.
- Ses paramètres sont :
 - La requête SQL
 - Le `RowMapper` à utiliser pour la création des objets
 - Un tableau de valeur pour remplacer les paramètres de la requête

```
@Override
public Client rechercherClientParId(long id) {
    String sql = "SELECT * FROM Client WHERE id=?";
    Client client = jdbcTemplate.queryForObject(
        sql, new ClientRowMapper(), new Object[] { id }
    );
    return client;
}
```

Spring JDBC – JdbcTemplate – Plusieurs résultats

- La récupération de plusieurs résultats se fait avec la méthode `query()` qui renvoi une implémentation de `java.util.List` d'objet associés.
- Pour permettre de typer correctement la collection renvoyée, il faut utiliser spécifier l'objet `RowMapper` associé à une représentation de la classe des objets.

```
@Override
public List<Client> rechercherTousLesClients() {
    String sql = "SELECT * FROM Client";
    List<Client> client = jdbcTemplate.query(
        sql, new ClientRowMapper()
    );
    return client;
}
```

Spring ORM – Utilisation avancée de la persistance

- Spring Framework est capable de prendre en charge l'intégration et l'utilisation des framework de mapping objet/relationnel (ORM) grâce à l'un de ses modules important : Spring ORM
- Spring ORM offre le support des frameworks suivant :
 - Hibernate
 - JPA (Java Persistence API)
 - Mybatis
- Dépendances :
 - `spring-orm`
 - Apport `spring-jdbc` et `spring-tx`
 - Ainsi que :
 - L'implémentation du framework de persistance à utiliser
 - Le pilote JDBC de la base de données sous-jacente

Intégration de JPA

- Il est possible d'utiliser JPA au travers Spring Framework.
 - Objectif : réduire la portion de code à réaliser.
- Spring Framework fournit pour cela :
 - La création, sous forme d'un bean, de la `EntityManagerFactory`
 - La configuration JPA est donc prise en charge par Spring
 - Plus de fichier `persistenc.xml`
 - Une classe de gestion de l'objet `EntityManager`
 - Il n'est plus nécessaire de créer et utiliser une classe utilitaire (EMFUtil...)
 - Une gestion simplifiée des transactions
 - Par annotations

Configuration de la EntityManagerFactory

- La EntityManagerFactory de JPA va être déclarée en tant que bean Spring.
 - Il est nécessaire d'utiliser, comme avec Spring JDBC, une implémentation de DataSource
 - Les propriétés peuvent également être externalisées dans un fichier *properties*

```
@Configuration
@ComponentScan({
    "fr.formation.banque"
})
@PropertySource({
    "classpath:/jpa.properties"
})
public class ConfigurationPersistence {

    @Autowired
    private Environment env;

    ...

}
```

```
@Bean("dataSource")
public DataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName(env.getProperty("db.driver"));
    dataSource.setUrl(env.getProperty("db.url"));
    dataSource.setUsername(env.getProperty("db.username"));
    dataSource.setPassword(env.getProperty("db.password"));
    return dataSource;
}

@Bean("entityManagerFactory")
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean entityManagerFactoryBean =
        new LocalContainerEntityManagerFactoryBean();

    entityManagerFactoryBean.setDataSource(dataSource());
    entityManagerFactoryBean.setPackagesToScan("fr.formation.banque.persistance.entity");

    HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    entityManagerFactoryBean.setJpaVendorAdapter(vendorAdapter);

    Properties props = new Properties();
    props.setProperty("hibernate.dialect", env.getProperty("hibernate.dialect"));
    props.setProperty("hibernate.show_sql", env.getProperty("hibernate.show_sql"));

    entityManagerFactoryBean.setJpaProperties(props);

    return entityManagerFactoryBean;
}
```

Utilisation de DAO JPA

- Spring Framework facilite la mise en œuvre de DAO avec JPA
- Une instance d'objet EntityManager peut être injectée dans les DAO grâce à l'annotation @PersistenceContext (javax.persistence)
- L'implémentation des méthodes se fait ensuite avec l'API native de JPA

```
@Component("clientDAO")
public class JpaClientDAO implements ClientDAO {

    @PersistenceContext
    private EntityManager em;

    ...
}
```

Les transactions

- Le conteneur Spring peut prendre en charge la gestion des transactions.
- Spring offre 2 approches pour gérer la démarcation des transactions :
 - démarcation par programmation
 - démarcation par déclaration
- Il faut privilégier la démarcation par déclaration car elle n'est pas intrusive pour le composant métier.
 - Le code des transactions est externalisé
- Pour gérer les transactions d'une API de persistance, il faut intégrer son gestionnaire de transaction en tant que bean dans la configuration

Le gestionnaire de transaction

- Il est déclaré dans la classe de configuration

```
@Bean("transactionManager")
public PlatformTransactionManager transactionManager(EntityManagerFactory emf) {
    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(emf);
    return txManager;
}
```

- Les transactions pourront être spécifiées par annotation
 - Il faut en activer le support !
 - Annotation `@EnableTransactionManagement` sur la classe de configuration

Utilisation des annotations de transaction

- L'annotation `@Transactional` va permettre de déclarer la stratégie transactionnelle des méthodes du DAO
 - L'annotation peut se positionner sur la classe, dans ce cas, toutes les méthodes adoptent le même comportement transactionnel.
 - `org.springframework.transaction.annotation.Transactional`
- Dans les DAO, l'intérêt est de positionner les annotations sur les interfaces, le comportement reste ainsi inchangé même si l'implémentation varie.

```
@Transactional(propagation = Propagation.REQUIRED, rollbackFor = BanqueException.class)  
public abstract void ajouterClient(Client client) throws BanqueException;
```

- Avec Java EE 7, une annotation `@Transactional` est apparu.

```
@Transactional(value=TxType.REQUIRED, rollbackOn=BanqueException.class)  
public abstract void ajouterClient(Client client) throws BanqueException;
```

Configuration des transactions

- Stratégies de propagation
 - **MANDATORY** : supporte la transaction courante; lève une exception s'il n'y a pas de transaction courante
 - **NESTED** : exécute dans une transaction si une transaction courante est créée
 - **NEVER** : ne supporte pas une transaction courante; lève une exception si une transaction courante existe
 - **NOT_SUPPORTED** : n'exécute jamais dans une transaction
 - **REQUIRED** : supporte la transaction courante et en crée une si elle n'existe pas
 - **REQUIRES_NEW** : crée une transaction et suspend la transaction courante si elle existe
 - **SUPPORTS** : supporte la transaction courante; si aucune transaction n'existe, ne crée pas de transaction

Bénéfices

- La configuration Java est réutilisable
 - On externalise les données de configuration changeante dans un fichier properties
- Le code des DAO JPA est allégé
 - Plus de gestion de l'objet EntityManager
 - Création, fermeture, ...
 - Plus gestion explicite des transactions
 - Démarrage implicite
 - Commit implicite
 - Rollback sur le(s) type(s) d'exception(s) déclarée(s) dans l'annotation
- Utilisation de l'API JPA native
 - Pas besoin de passer par une couche d'abstraction

Travaux Pratiques



www.eni-service.fr



Module 5

Les tests avec Spring

Contenu du module

- Tests avec Spring
- Configuration des tests
- Spécificités dans les tests

Les tests avec Spring

- Les tests d'intégration tiennent compte des rôles de Spring Framework
 - Avec un contexte d'application éventuellement réduit (ou adapté), comprenant uniquement un ensemble de classes que l'on veut tester
 - Avec une configuration d'infrastructure spécifique :
 - Une base de données en mémoire
 - Une instance spécifique de la base de données cible
- Spring propose une intégration à JUnit qui simplifie grandement ce type de configuration
 - Les alternatives à Spring ont généralement des mécanismes similaires
 - Cas d'une application Full-Java EE par exemple
- Dépendances :
 - `spring-test`

Support des tests

- Le « runner » spécifique **SpringJUnit4ClassRunner** permet d'intégrer Spring dans un test JUnit
- L'annotation **@ContextConfiguration** permet alors de localiser la configuration de Spring et d'amorcer le contexte d'application
 - Et donc de choisir une configuration particulière pour les tests !
- On peut ainsi tester son application Spring avec JUnit, sans serveur d'applications
 - Toutes les fonctionnalités gérées par Spring fonctionnent
 - Connexion à une base de données
 - Transactions...

Spring et les tests : A savoir

- Spring ne lance qu'un seul contexte d'application par classe !
 - Toutes les méthodes de test d'une classe donnée utilisent la même instance
 - L'objectif est d'accélérer les tests
- Par défaut, toute méthode de test annotée avec **@Transactional** va être annulée (rollback) à la fin du test
 - Il est donc inutile de prévoir de nettoyer la base de données après un test
 - Le test sera également plus performant
 - A condition que personne ne fasse de « commit » explicitement pendant le test !

Mise en œuvre sur un DAO

■ Une classe de test JUnit

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = ConfigurationPersistence.class)
@Transactional
public class JpaClientDAOTest {

    @Autowired
    private ClientDAO clientDAO;

    @Test
    public void testAjouterClient() {
        Client client = new Client();
        client.setPrenom("G rard");
        client.setNom("LEPIC");
        client.setAdresse("56 rue de Nantes");
        client.setCodePostal("35000");
        client.setVille("Rennes");
        client.setMotDePasse("123456");

        clientDAO.ajouterClient(client);

        List<Client> listeDesClients = clientDAO.rechercherTousLesClients();
        assertEquals(5, listeDesClients.size());
    }

    ...
}
```

Test g r  par Spring
Configuration
On annule les
modifications en base
de donn es apr s
les tests

On injecte le composant   tester

M thode de test JUnit

V rification...

Travaux Pratiques



www.eni-service.fr



Module 6

Les applications Web - Spring MVC

Contenu du module

- Présentation de Spring MVC
 - Principes de Spring MVC
 - Le design-pattern MVC 2
 - Implémentation de MVC 2 dans Spring
 - Les objets MVC 2 dans Spring MVC
- Configuration d'une application Web Spring MVC
 - Initialisation du framework Spring MVC
 - Les contextes
 - Le contrôleur frontal
 - Les annotations
- Les contrôleurs
 - Le traitement des requêtes par les contrôleurs
 - Les contrôleurs basiques
 - Les contrôleurs avancés
 - La gestion des formulaires
- Les vues
 - La gestion des vues dans Spring MVC
 - La configuration de la vue
 - Les différentes implémentations
 - Cas particulier des vues de formulaire

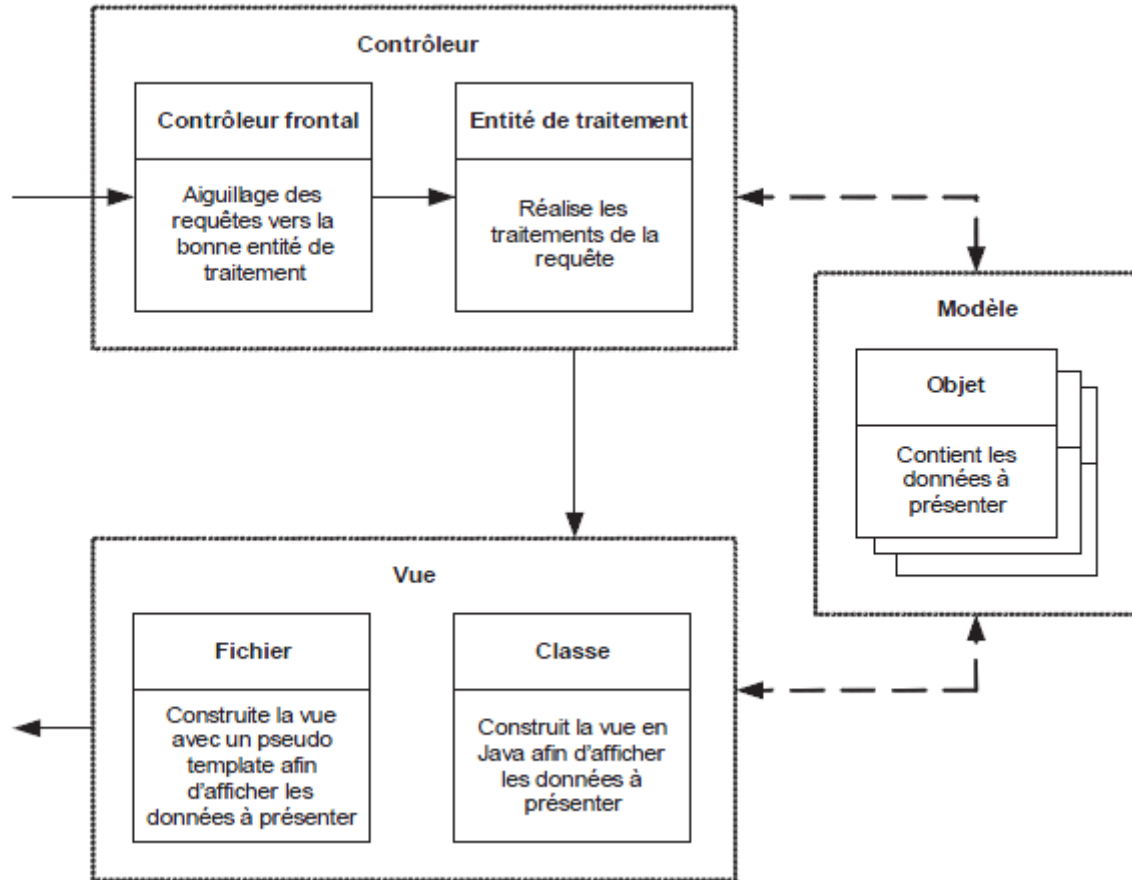
Spring MVC

- Spring MVC est un framework de présentation pour le développement d'applications Web Java EE, conforme au modèle de conception MVC, et fondé sur le conteneur léger de Spring
- Dans le cas de Spring MVC, le conteneur va servir à créer :
 - Le contexte de l'application Web
 - Les objets traitant les requêtes (Controller)
 - Les objets créant les pages HTML (View)
 - Les objets données des formulaires (Command)
 - Les liens avec les couches métiers et d'accès aux données
 - Et pleins d'autres ...
 - Le mapping des URL vers les contrôleurs
 - Le mapping des vues , etc.
- Dépendances :
 - `spring-webmvc`

MVC et Spring

- Le patron MVC est communément utilisé dans les applications Java/Java EE pour réaliser la couche de présentation des données aussi bien dans les applications Web que pour les clients lourds.
- Lorsqu'il est utilisé dans le cadre de Java EE, il s'appuie généralement sur l'API servlet ainsi que sur des technologies telles que JSP/JSTL.
- Il existe deux types de patrons MVC :
 - Le type 1, qui possède un contrôleur par action,
 - Le type 2, plus récent et plus flexible, qui possède un contrôleur unique.
- C'est ce dernier qui est implémenté dans les frameworks MVC, donc Spring MVC.

Organisation du pattern MVC 2



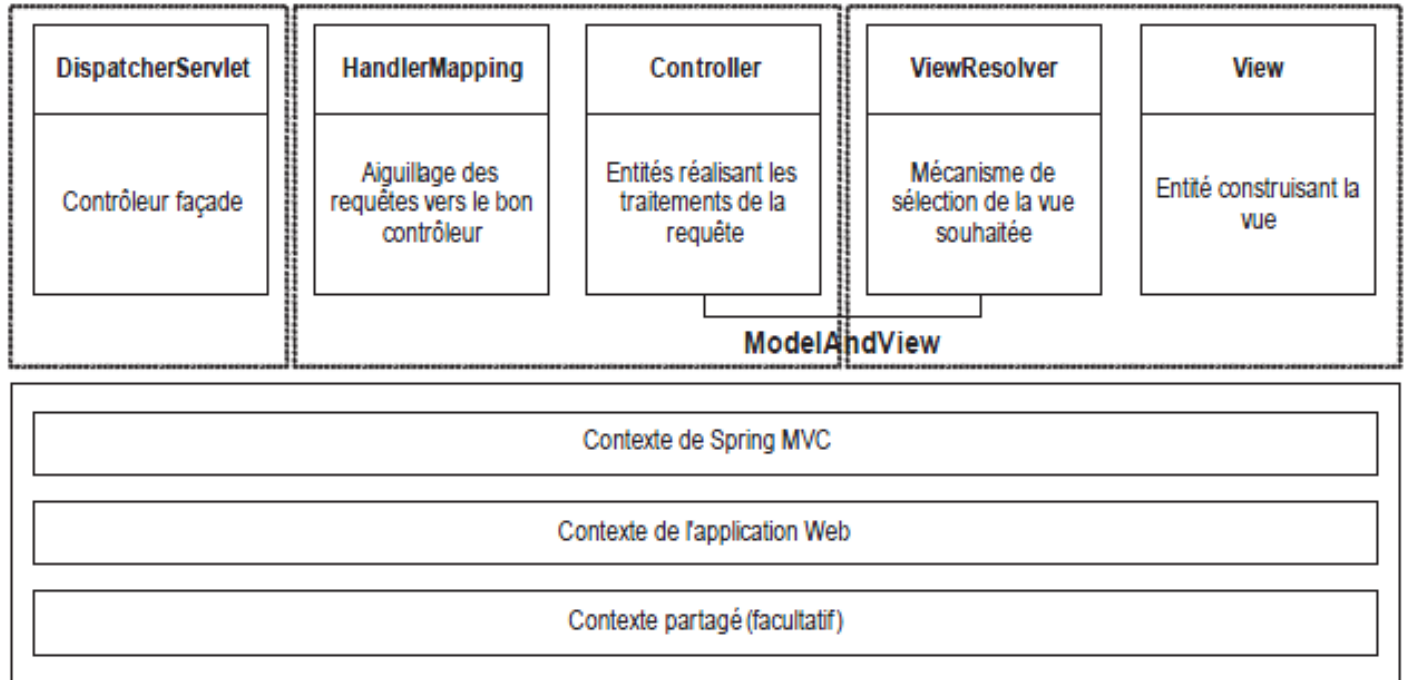
Organisation du pattern MVC 2 (suite)

- Les composants mis en œuvre dans ce pattern sont les suivantes :
 - Modèle. Permet de mettre à disposition les informations utilisées par la suite lors des traitements de présentation. Cette entité est indépendante des API techniques et est constituée uniquement de Beans Java.
 - Vue. Permet la présentation des données du modèle. Il existe plusieurs technologies de présentation, parmi lesquelles JSP/JSTL, XML, les moteurs de templates Velocity et Free-Marker ou de simples classes Java pouvant générer différents types de formats.
 - Contrôleur. Gère les interactions avec le client tout en déclenchant les traitements appropriés.
- Le contrôleur a un rôle fondamental, il interagit directement avec les composants de la couche service métier et a pour responsabilité la récupération des données mises à disposition dans le modèle.
- Dans la mise en œuvre de MVC 2, le contrôleur se compose d'un point d'entrée unique pour toute l'application (Front Controller) et de plusieurs actions de traitement.

Implémentation du pattern MVC 2 dans Spring

- La configuration des contrôleurs MVC se réalisent par l'intermédiaire d'annotations.
- Grâce à elles, Spring MVC permet de masquer l'utilisation de l'API servlet et favorise la mise en œuvre des tests unitaires à ce niveau.
- La gestion des formulaires se fait également grâce aux annotations. Elle permettent non seulement de charger et d'afficher les données du formulaire, mais également de gérer leur soumission.
 - Ces données sont utilisées pour remplir directement un Bean sans lien avec Spring MVC, (il peut être validé éventuellement).
 - Des mécanismes de mappage et de conversion des données sont disponibles.
- Spring MVC offre enfin des possibilités d'abstraction de la technologie d'implémentation des vues, permettant de changer de technologie de présentation sans impacter le contrôleur.

Les objets d'implémentation



Initialisation du framework Spring MVC

- L'initialisation du framework Spring MVC s'effectue en deux parties, essentiellement au sein du fichier `web.xml`.
- Dans le cadre d'applications Java EE, une hiérarchie de contextes est mise en œuvre afin de regrouper et d'isoler de manière logique les différents composants
- Spring framework offre une hiérarchie pour les deux contextes suivants :
 - **Contexte de l'application Web (contexte racine)**. Stocké dans le `ServletContext`, ce contexte doit contenir la logique métier ainsi que celle de l'accès aux données.
 - **Contexte du framework MVC**. Géré par le contrôleur façade du framework, ce contexte doit contenir tous les composants relatifs au framework MVC utilisé.

Chargement du contexte de l'application Web

- Le framework Spring fournit une implémentation de « listener » Java EE permettant de configurer et d'initialiser le contexte au démarrage et de le finaliser à l'arrêt de l'application Web.
 - La déclaration de ce listener est à ajouter au fichier `web.xml`.
- Il permet de démarrer le conteneur léger et est donc nécessaire à toutes applications Web Java EE utilisant Spring

```
<context-param>
  <param-name>contextClass</param-name>
  <param-value>
    org.springframework.web.context.support.AnnotationConfigWebApplicationContext
  </param-value>
</context-param>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    fr.formation.banque.config.ConfigurationPersistance,
    fr.formation.banque.config.ConfigurationMetier
  </param-value>
</context-param>
```

Chargement du contexte de Spring MVC

- La configuration et le chargement de ce contexte sont liés à ceux de la servlet du contrôleur frontal de Spring MVC
- La configuration consiste donc à déclarer cette servlet, l'associer à un motif d'URL, la charger au démarrage et lui donner la configuration Spring MVC

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextClass</param-name>
    <param-value>
      org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
  </init-param>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>fr.formation.banque.config.ConfigurationWeb</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Les annotations

- La configuration des contrôleurs se réalise par l'intermédiaire d'annotations.
 - Il est nécessaire d'activer leurs support dans la configuration de Spring.
- Deux éléments de configuration sont nécessaires :
 - L'annotation `@EnableWebMvc` pour activer le support des annotations de contrôleur
 - L'annotation `@ComponentScan` pour indiquer les packages à analyser

```
@Configuration
@EnableWebMvc
@ComponentScan({
    "fr.formation.banque.Web"
})
public class ConfigurationWeb {

}
```

Le traitement des requêtes par les contrôleurs

- Pour le traitement des requêtes, un mécanisme de correspondance entre la classe de traitement et l'URI de la requête est fourni.
- La correspondance est réalisé par les annotations RequestMapping des contrôleurs.
- L'URL d'une requête, apparaît toujours sous la forme suivante dans les applications Java EE :
 - `http://<machine>:<port>/<alias-webapp>/<alias-ressource-web>`
- Il est possible d'utiliser l'annotation au niveau de la classe de contrôleur mais également au niveau des méthodes de ce contrôleur.

```
@Controller
@RequestMapping("/welcome.do")
public class WelcomeController {
    public void welcome() {
        ...
    }
}
```

```
@Controller
public class WelcomeController {
    @RequestMapping("/welcome.do")
    public void welcome() {
        ...
    }
}
```

Configuration avancée

```
@Controller
```

```
public class WelcomeController {  
    @RequestMapping(  
        value={"/welcome.do", "/index.do"},  
        method=RequestMethod.GET  
    )  
    public void welcome() {  
        ...  
    }  
}
```

Plusieurs mappages

Spécification de
méthode HTTP

Les contrôleurs avancés

- Les méthodes de gestion d'un contrôleur ne sont pas tenues de respecter la signature héritée de l'API Servlet (`HttpServletRequest` et `HttpServletResponse`)
- Pour faciliter le traitement du passage des paramètres de requête, et éviter d'utiliser les objets `HttpServletRequest` et `HttpServletResponse`, Il est possible d'utiliser l'annotation `@RequestParam`
- Pour simplifier la manipulation du modèle et l'invocation de la vue, un paramètre de type `ModelMap` pourra être passé.

Les contrôleurs avancés : paramètres de requête

- L'annotation `@RequestParam` offre la possibilité de référencer un paramètre de la requête par son nom. L'objet correspondant est alors passé en tant que paramètre.

- Une conversion automatique de type est réalisée si nécessaire.

`@Controller`

```
public class WelcomeController {  
    @RequestMapping("/welcome.do")  
    public String welcome(  
        @RequestParam String listId) throws Exception {  
        ...  
    }  
}
```

- Il est possible de spécifier le nom du paramètre de requête (si différent de celui de la méthode), ainsi que d'imposer sa présence

`@Controller`

```
public class WelcomeController {  
    @RequestMapping("/welcome.do")  
    public String welcome(  
        @RequestParam(value="listId", required="false") String listId  
    ) throws Exception {  
        ...  
    }  
}
```


Les contrôleurs avancés : ModelMap

- Les méthodes de traitement des contrôleurs acceptent un paramètre de type ModelMap, ce paramètre correspondant aux données du modèle.
- En utilisant ce paramètre, il est possible de manipuler les données du modèle et d'en ajouter de nouvelles.
- Une chaîne de caractères correspondant à l'identifiant de la vue suffit.

@Controller

```
public class WelcomeController {  
    @RequestMapping("/welcome.do")  
    public String welcome(@RequestParam String listId,  
        ModelMap model) throws Exception {  
        ...  
        model.addAttribute("defaultList", aList);  
        return "welcomes";  
    }  
}
```

La gestion des formulaires

- Spring MVC fournit un support pour l'affichage des données des formulaires et leur soumission à l'aide d'annotations.
- Les contrôleurs de formulaire utilisent un Bean afin de stocker les informations des formulaires, aucune configuration n'est à réaliser pour l'injection de dépendances.
- Il suffit que ce Bean soit présent dans les données du modèle et que l'identifiant correspondant soit spécifié dans le formulaire.

Affichage d'un formulaire

- L'utilisation des annotations `@RequestMapping`, `@ModelAttribute` et `@InitBinding` permet de charger les différentes entités nécessaires à l'affichage du formulaire dans la vue.
- L'affichage du formulaire est réalisé grâce à l'appel d'une méthode de traitement de la requête par le biais de la méthode GET.
- Spring MVC permet d'initialiser l'objet de formulaire en se fondant sur une méthode annotée par `@ModelAttribute`.
- Cet objet doit être retourné par la méthode et est automatiquement ajouté dans le modèle. Il peut donc être utilisé par la suite dans la vue pour initialiser le formulaire correspondant.

```
public class FormController {  
    ...  
    @ModelAttribute("userinfo")  
    public UserInfoData initForm(HttpServletRequest request) {  
        ...  
        UserInfoData data = new UserInfoData();  
        ...  
        return data;  
    }  
    ...  
}
```

Affichage d'un formulaire (suite)

- Ensuite, il faut définir une méthode de traitement dédiée à l'affichage du formulaire. Elle doit être annotée avec `@RequestMapping` et posséder la propriété `method` avec la valeur `RequestMethod.GET`.
 - Le mappage avec l'URI peut être spécifié à ce niveau ou globalement au niveau de la classe.
- Cette méthode ne possède pas particulièrement de traitements, mais spécifie la vue correspondant au formulaire.

```
public class FormController {  
    ...  
    @RequestMapping(method=RequestMethod.GET)  
    public String showForm() {  
        return "userinfo";  
    }  
    ...  
}
```

Soumission d'un formulaire

- La méthode de traitement dédiée à la soumission du formulaire doit être annotée avec `@RequestMapping` et posséder la propriété `method` avec la valeur `RequestMethod.POST`.
- Cette méthode prend en paramètre l'objet de formulaire, objet annoté par `@ModelAttribute` et a la responsabilité de traiter cet objet.

...

```
@RequestMapping(method = RequestMethod.POST)
public String submitForm(
    @ModelAttribute("userinfo") UserInfoData userInfo)
{
    return "userinfoview";
}
```

La gestion de la vue

- Utilisation de la classe `ModelMap` en tant que paramètre d'une méthode de traitement annotée avec `@RequestMapping`.
- Ce paramètre correspond à l'entité de stockage des éléments du modèle
- Il faut spécifier l'identifiant de la vue choisie en le faisant retourner sous forme de chaîne de caractères par la méthode.

`@Controller`

```
public class WelcomeController {  
    @RequestMapping("/welcome.do")  
    public String welcome(@RequestParam String listId,  
        ModelMap model) throws Exception {  
        ...  
        return "welcomes";  
    }  
}
```

Identification de la vue

Configuration de la vue

- La sélection des vues dans Spring MVC est effectuée par le biais d'une implémentation de l'interface `org.springframework.web.servlet.ViewResolver`.
- Trois implémentations principales :
 - `ResourceBundleViewResolver`, correspond à une configuration des vues dans un fichier properties.
 - `XmlViewResolver`. Les vues sont définies dans un sous-contexte de Spring. L'utilisation de toutes les fonctionnalités et mécanismes du framework est donc envisageable, de même que l'injection de dépendances sur la classe d'implémentation des vues.
 - `InternalResourceViewResolver` utilise les URI dans le but de résoudre les vues. Ce mécanisme construit l'URI à partir de l'identifiant de la vue.

Configuration du Resolver

- Dans la classe de configuration du framework MVC
- Ici, si un contrôleur retourne la chaine `welcomes` alors la vue localisée sera `/WEB-INF/jsp/welcomes.jsp`

```
@Configuration
@EnableWebMvc
@ComponentScan("fr.formation.banque.web")
public class ConfigurationWeb {

    @Bean
    public ViewResolver configureViewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();

        viewResolver.setPrefix("/WEB-INF/jsp/");
        viewResolver.setSuffix(".jsp");

        return viewResolver;
    }
}
```


Réalisation des vues

- Spring MVC fournit une vue fondée sur JSP/JSTL : `JstlView`.
- Les balises et expressions JSTL peuvent être utilisées d'une manière classique en utilisant les données du modèle.
- Afin d'utiliser les taglibs JSTL, des importations doivent être placées dans les pages JSP (les JARs de la JSTL doivent être présents).

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
```

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt_rt" %>
```

- Au niveau des formulaires, un taglib dédié permet de réaliser le mappage entre les données du formulaire et les champs correspondants.
- Pour pouvoir l'utiliser, l'importation suivante doit être placée dans les pages JSP :

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

Cas spécifique : Les vues de formulaire

- L'utilisation des balises du taglib form permet d'initialiser automatiquement les champs du formulaire avec les données du formulaire et d'afficher les éventuelles erreurs survenues.

```
<form:form modelAttribute="userinfo">
<tr>
  <td>
    <form:input path="firstName" size="15" maxLength="60" />
  </td>
</tr>
...
</form:form>
```

Travaux Pratiques





Module 7

Services Web – Spring Rest

Contenu du module

- Présentation de Spring Rest
- Configuration
- Les contrôleurs Rest
- Les formats de données
- Récupération des données
 - Dans l'URL
 - Dans la requête
- Gestion de la réponse
- Gestion des exceptions

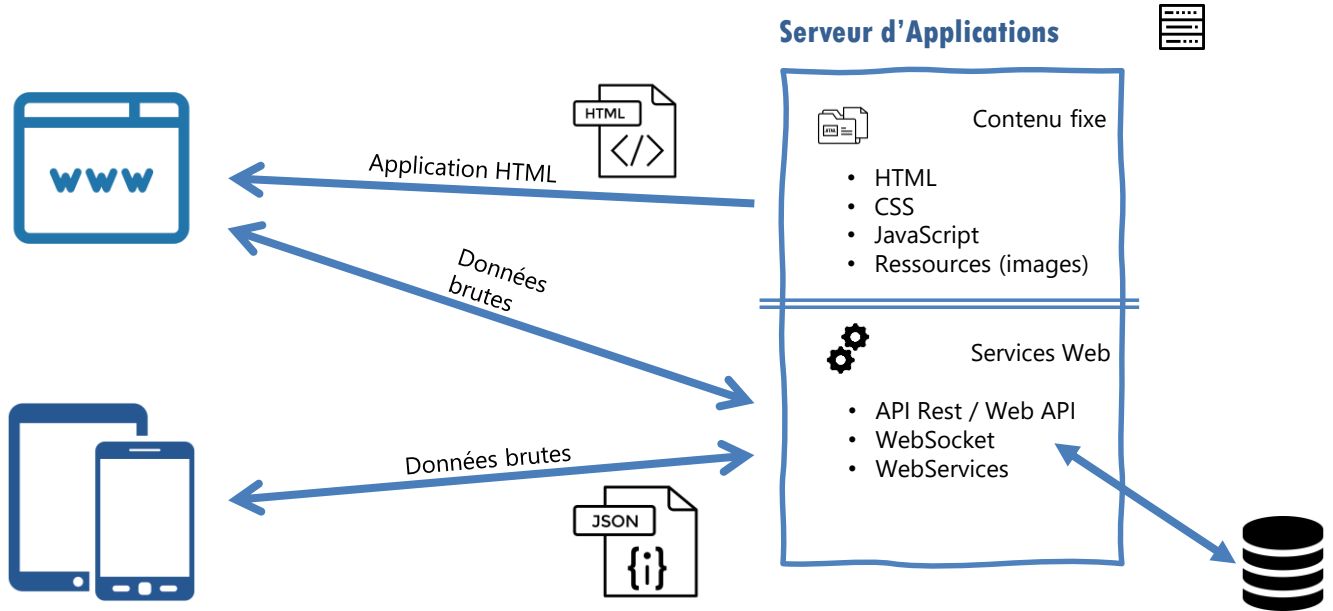
Principes de REST

- Cette architecture part du principe selon lequel Internet est composé de ressources accessibles à partir d'une URL.
 - Par exemple, pour avoir le temps à Paris, un utilisateur pourrait utiliser une adresse de la forme <http://www.meteo.fr/paris> : Paris serait alors une ressource telle que définie par Météo France.
- A la requête de cet URL serait renvoyée une représentation de la ressource demandée (ex : paris.jsp). Cette représentation place l'application cliente dans un état (*state*) donné.
- Si l'application cliente lance un appel sur un des liens de la représentation en cours, une autre ressource est appelée, dont une représentation est envoyée. Ainsi, l'application cliente change d'état (*state transfer*) pour chaque représentation de ressource.

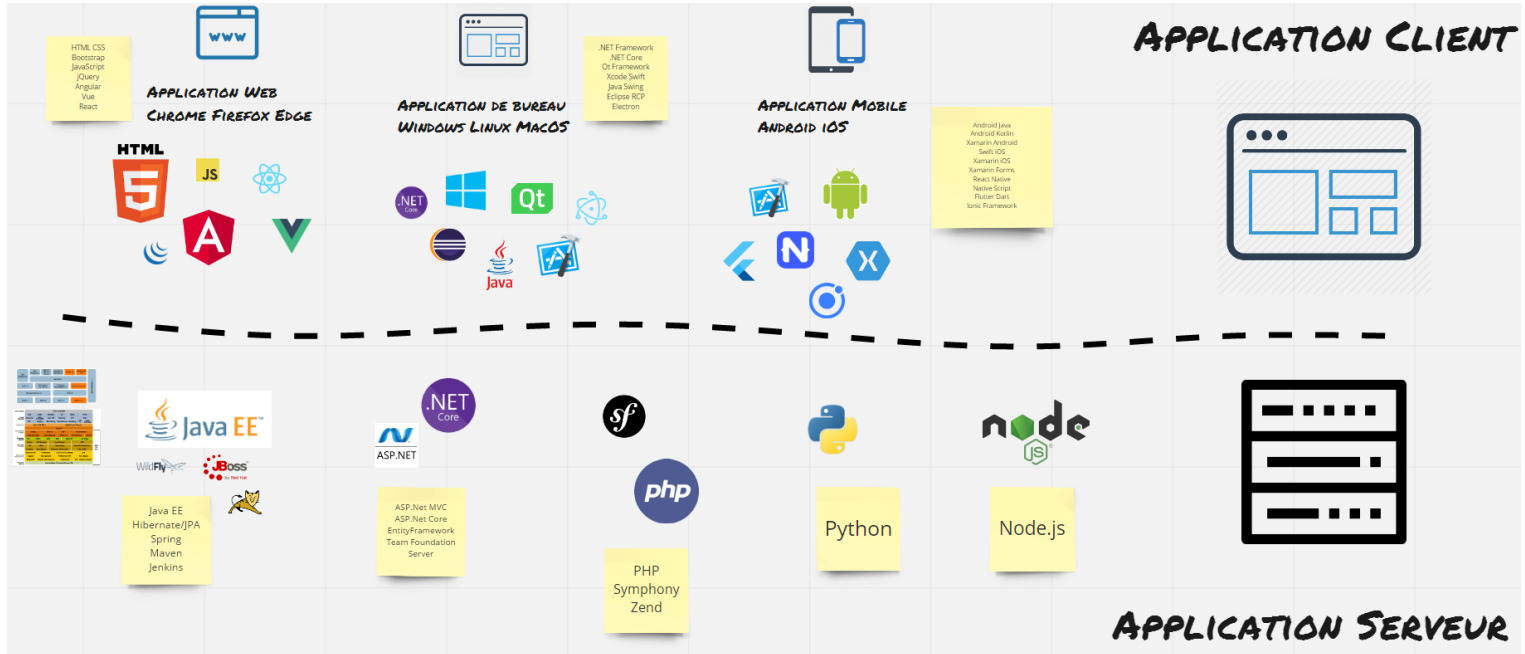
Caractéristiques de REST

- Utilise les standards de l'Internet
 - Architecture orientée ressource basée sur des URI (*Uniform Resource Identifier*)
 - Repose sur l'utilisation du protocole HTTP et de ses méthodes (POST, GET, PUT, DELETE)
- Echange de données multi-formats
 - en XML ou autres (JSON, objets sérialisés en binaire)
- Utilisation des standards hypermedia : HTML ou XML qui permettent de faire des liens vers d'autres ressources et d'assurer ainsi la navigation dans l'application REST
- Utilise les types MIME pour la représentation des ressources (text/xml, image/jpeg, application/pdf, text/html, ...)
- Implémentation restreinte : il faut comprendre l'architecture REST et ensuite concevoir des applications ou des services Web selon cette architecture
- Alternative à SOAP censée être plus simple
- Aucun langage standard de description de service
- N'est pas un standard

Architecture REST



Stacks techniques



Principe de fonctionnement de REST

- Les méthodes du protocole HTTP servent à définir le type traitement à effectuer

Action	SQL	HTTP
Create	Insert	POST
Read	Select	GET
Update	Update	PUT
Delete	Delete	DELETE

- Donc pour l'URI [/client/123](#), l'action sur cette ressource est décidée par la méthode HTTP utilisée.
 - Ce ne sont cependant que des conventions !
 - L'implémentation est à réaliser par le programmeur.

Règles de conception d'un Service REST

- Toutes les ressources devant être exposées au travers du service doivent être correctement identifiées, et de manière unique. Chaque ressource devra se voir assigner une URL.
 - Qui plus est, l'URL en question devra être de la forme <http://www.site.com/contenus/003> plutôt que <http://www.site.com/contenus?id=003>.
- Les ressources doivent être catégorisées selon leurs possibilités offertes à l'application cliente
 - Ne peut-elle que recevoir une représentation (GET) ?
 - Ou bien peut-elle aussi modifier/créer une ressource (POST, PUT, DELETE) ?
- Chaque ressource devrait faire un lien vers les ressources liées
- La manière dont fonctionne le service sera décrite au sein d'un simple document HTML
 - Qui servira d'interface et de documentation d'API

JSON

- Le format JSON (*JavaScript Object Notation*) est un format de données textuel inspiré de la notation des objets JavaScript créé à partir de 2002 par Douglas Crockford.
- L'objectif du format JSON est de faciliter les échanges de données applicatives sur le Web qui étaient auparavant systématiquement basés sur XML
 - Jugé trop verbeux et donc consommateur de bande passante, JSON constitue une alternative plus légère pour ces échanges.

Représentation des données en JSON

- JSON représente les données sous forme d'objets, ils sont constitués d'attributs exprimés grâce à des paires clé/valeur, la clé correspondant au nom de l'attribut et la valeur à la donnée associée.
 - Cette donnée peut être un numérique, une chaîne de caractères, un autre objet ou bien un tableau.
- Les clés sont des chaînes de caractères et il est nécessaire de les exprimer entre guillemets.
- Les valeurs sont séparées des clés par le caractère deux points (:) et chaque couple clé/valeur est séparé du suivant par une virgule (,).
- Exemple de structure :
...
`"clé1": "Valeur de clé1",`
`"clé2": "Valeur de clé2"`
...

JSON : Type de données

- Chaines de caractères
 - Elles sont exprimées en Unicode et entre guillemets ;
- Les numériques
 - Entiers ou nombres décimaux ;
- Les booléens
 - Exprimés avec les mots réservés **true** et **false** ;
- Les objets
 - La base de la notation de JSON ;
- Les tableaux
 - Des ensembles de valeurs des types précédents.

Structures JSON

- Les paires de clé/valeur de JSON sont exprimés entre accolades ({}), c'est la manière de représenter un objet. Ainsi si l'on souhaite exprimer les données d'un objet **personne** qualifié par un **nom**, un **prénom** et un **âge**, on pourrait utiliser la structure suivante :

```
{  
    "nom": "DUPONT",  
    "prenom": "Robert",  
    "age": 56  
}
```

- Dans cet exemple, les valeurs pour le nom et le prénom sont des chaînes de caractères, l'âge est un entier.
- Les tableaux sont quant à eux exprimés entre crochets ([]), par exemple :

...

```
"clé1": [ "Première valeur de clé1", "Deuxième valeur de clé1" ]
```

...

Spring Rest

- Spring Rest est un sous-ensemble de Spring MVC qui permet la réalisation de contrôleur exposant des APIs Restful
- En terme de dépendance de librairie, on utilise celle de Spring MVC
- Spring Rest apporte essentiellement des annotations spécifique permettant d'associer les méthodes Java des contrôleurs à des URLs et des verbes HTTP
- Annotations :
 - Sur les classes :
 - @RestController
 - Sur les méthodes :
 - @GetMapping, @PostMapping, @PutMapping, @DeleteMapping
 - Pour récupérer les données :
 - @RequestBody, @PathVariable
- NOTE : Spring Rest n'est pas une implémentation de la sépcification JAX-RS, il s'agit d'une API propre à Spring

Configuration de Spring Rest

- Il est nécessaire de disposer :
 - D'une classe de configuration
 - Qui active le support WebMVC et l'analyse des packages à la recherche de classes annotées
 - De la configuration permettant de charger le contrôleur frontal dans le fichier `web.xml`
- Classe de configuration :

```
@Configuration
@EnableWebMvc
@ComponentScan({
    "fr.formation.banque.rest"
})
public class ConfigurationRest {

}
```

Chargement du contrôleur frontal Rest

- L'ensemble des requêtes http://.../api/* seront dirigées vers les contrôleurs Rest
- Les classes de contrôleurs seront stockées dans le packages indiqué précédemment avec le @ComponentScan

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextClass</param-name>
    <param-value>
      org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
  </init-param>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>fr.formation.banque.config.ConfigurationRest</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/api/*</url-pattern>
</servlet-mapping>
```

Les contrôleurs Rest

- Les contrôleurs Rest sont des classes Java annotées avec :
 - `@RestController`
 - Leur donne cette « nature » spécifique permettant leur gestion par le framework MVC
 - `@RequestMapping`
 - Pour spécifier un préfix d'URI utilisé pour identifier les fonctionnalités de ce contrôleur

```
@RestController
@RequestMapping("/api")
public class BanqueRestService {

}
```

- Ils contiendront des méthodes dont l'invocation sera déclenchée par une requête HTTP
 - Les annotations `@GetMapping`, `@PutMapping`, `@PostMapping` et `@DeleteMapping` permettent de spécifier l'URI

Le mapping des URLs

- Les annotations de mappings se posent sur les méthodes des contrôleurs Rest

```
@PostMapping(value = "/auth", consumes = "application/json", produces = "application/json")  
public Client authentifier( ... ) { ... }
```

- Chaque annotation permet de spécifier :
 - La méthode HTTP d'invocation, c'est le nom de l'annotation
 - L'URI, avec l'attribut value
 - Le format de donnée consommé, avec l'attribut consumes
 - Le format de données produit, avec l'attribut produces

Gestion des formats de données

- En utilisant `consumes` et `produces`, on spécifie les types de données attendus et produits. Encore faut-il que la classe Java soit capable de faire la transformation !
 - On a besoin d'outils de sérialisation/désérialisation !
 - Ils vont permettre de réaliser les conversions Java vers le format, et du format vers Java
- Sérialiseur/Désérialiseur :
 - Format Texte brut (text/plain) : Chaîne de caractères Java
 - Format XML (text/xml) : API JAXB, natif dans Java depuis Java 6
 - Format JSON (application/json) :
 - Pas natif en Java ! Besoin d'utiliser une bibliothèque tierce

Gérer le format JSON

- Jackson est un projet Open Source hébergé sur GitHub implémentant une prise en charge du format JSON pour Java
 - <https://github.com/FasterXML/jackson>
- La mise en œuvre de Jackson nécessite d'en déclarer les librairies en tant que dépendance du projet

Récupérer les données transmises

- L'URI d'un service REST peut transporter des informations importantes pour l'invocation du service -> Seule solution en GET !
 - Exemple : Un identifiant d'article
 - /api/article/**568942**
- Avec Spring Rest, on peut simplement associer une portion de l'URI avec un paramètre de la méthode Java invoquée grâce à l'annotation `@PathVariable`

```
@GetMapping(value = "/article/{id}", produces = "application/json")  
public Article getArticle(  
    @PathVariable("id") long idArticle  
) {  
  
    ...  
}
```

- Une conversion automatique est faite si nécessaire

Travailler avec la requête

- Dans le cas de requêtes PUT et POST, il est courant de transporter des informations dans le corps de la requête.
- Ces informations doivent pouvoir être désérialisées dans des objets Java
- On peut associer la structure d'un flux (JSON par exemple) transporté dans le corps de la requête à la structure d'un objet Java grâce à l'annotation `@RequestBody`

```
@PostMapping(  
    value = "/auth",  
    consumes = "application/json",  
    produces = "application/json"  
)  
public Client authentifier(  
    @RequestBody Auth auth  
) {  
    ...  
}
```

Le contenu attendu dans le corps de la requête est un flux JSON

Le flux sera automatiquement désérialisé dans un objet de type Auth (Classe de l'application)

Travailler avec la réponse

- Il est parfois nécessaire de contrôler la manière dont la réponse est générée et dans ce cas, se contenter de renvoyer un objet sérialisé ne suffit pas.
 - Il peut être par exemple nécessaire d'agir sur le code de réponse HTTP
- Il est possible de créer et renvoyer des objets de type `org.springframework.http.ResponseEntity<T>` où T représente le type de l'objet sérialisé dans la réponse

```
@GetMapping(value = "/article/{id}", produces = "application/json")
public ResponseEntity<Article> getArticle(
    @PathVariable("id") long idArticle
) {
    ...
}
```

Construire la réponse

- Pour construire la réponse, il suffit d'instancier la classe `ResponseEntity` et de renvoyer l'objet
- Constructeurs :
 - `ResponseEntity(HttpStatus status)`
 - Create a `ResponseEntity` with a status code only.
 - `ResponseEntity(MultiValueMap<String,String> headers, HttpStatus status)`
 - Create a `ResponseEntity` with headers and a status code.
 - `ResponseEntity(T body, HttpStatus status)`
 - Create a `ResponseEntity` with a body and status code.
 - `ResponseEntity(T body, MultiValueMap<String,String> headers, HttpStatus status)`
 - Create a `ResponseEntity` with a body, headers, and a status code.
 - `ResponseEntity(T body, MultiValueMap<String,String> headers, int rawStatus)`
 - Create a `ResponseEntity` with a body, headers, and a raw status code.

Gestion des erreurs avec Spring Rest

- Les exceptions doivent être remontées sous forme de réponse HTTP avec les bons codes de réponse
- On pourrait renvoyer un objet `ResponseEntity` avec un flux de message et le code de réponse approprié mais cela nécessiterait de créer des objets supplémentaires...
 - De plus, cela ne représente pas réellement la notion d'erreur dans le flux du programme...
- Solution : Générer une exception fournie par Spring Rest
 - `ResponseStatusException`
 - Elle encapsule la notion de message d'erreur et de code de réponse HTTP
 - C'est une *unchecked exception* donc il n'est pas nécessaire d'en déclarer le déclenchement dans la signature de la méthode

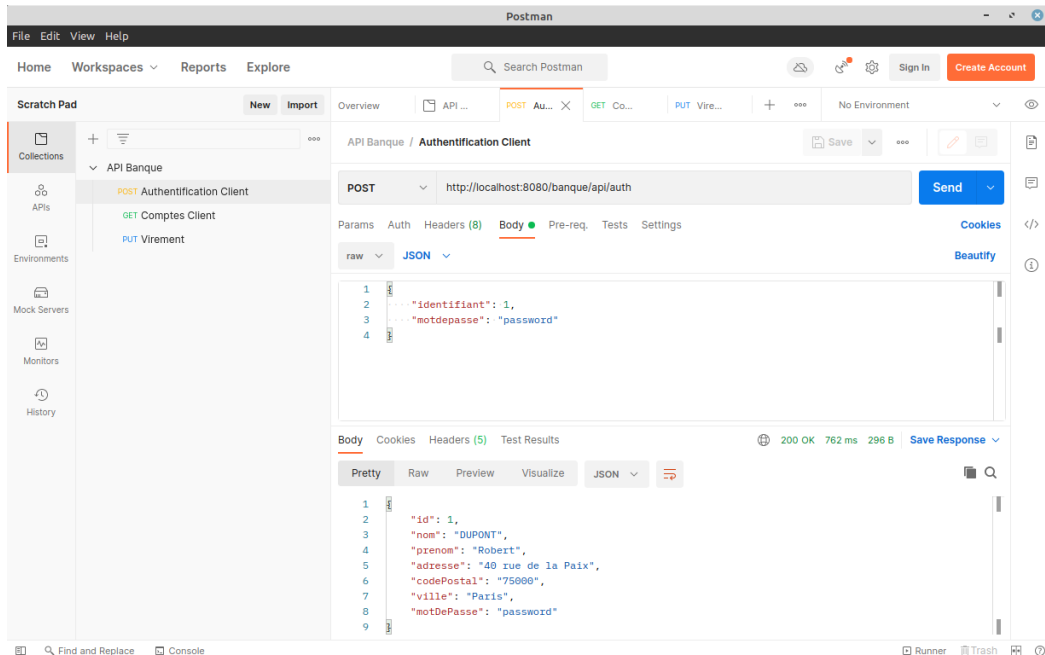
Utiliser ResponseStatusException

- Un type d'exception adapté aux réponses HTTP
- Constructeurs :
 - `ResponseStatusException(HttpStatus status)`
 - `ResponseStatusException(HttpStatus status, String reason)`
 - `ResponseStatusException(HttpStatus status, String reason, Throwable cause)`
 - `ResponseStatusException(int rawStatusCode, String reason, Throwable cause)`

```
catch (MyApplicationException e) {  
    throw new ResponseStatusException(  
        HttpStatus.INTERNAL_SERVER_ERROR,  
        e.getMessage()  
    );  
}
```

Tester son API Rest

- Pour tester une API Rest il est indispensable de disposer d'un outil capable d'émettre des requêtes HTTP et de collecter les réponses
- Outil : Postman
 - <https://www.postman.com>



Travaux Pratiques



www.eni-service.fr



Module 8

Intégration avec JMS

Contenu du module

- Les technologies d'intégration
 - Les technologies d'intégration de service dans Java / Java EE
 - Support de ces technologies dans Spring Framework
- La messagerie applicative JMS
 - La messagerie asynchrone Java
 - Utilisation de JMS avec Spring Framework
 - Configuration des fabriques et destinations avec Spring
 - La classe JmsTemplate

La problématique : EIS et EAI

- Toutes les applications métiers d'entreprise doivent pouvoir s'intégrer dans les systèmes d'information des entreprises : EIS (Enterprise Information System)
- Elles doivent pouvoir réutiliser des services applicatifs existants, tout en minimisant les duplications de données dans ces différents systèmes.
 - Principes de base de l'urbanisation des systèmes d'information
- L'interaction entre des applications pouvant être séparées physiquement au sein de l'entreprise et utilisant des mécanismes ou des technologies hétérogènes peut vite devenir complexe, puisqu'il n'est pas toujours possible de les réécrire ou de les modifier.
 - Cette réécriture n'est pas forcément la meilleure solution pour des applications répondant aux besoins et fonctionnant correctement.
 - L'interaction avec elles est la solution la plus adaptée
- Cette interaction peut s'insérer dans différents types de traitements et mettre en œuvre des mécanismes de communication complexes, synchrones ou asynchrones appropriée.

Les technologies d'intégration de service dans Java / Java EE

- La plateforme Java, et plus spécifiquement, la plateforme Java EE, est réputée pour sa robustesse mais également pour ses capacités à fournir des solutions d'intégration de services applicatifs hétérogènes.
 - 62% de la motivation de l'adoption de Java EE pour les applications métier (France, 2010)
- Il est facile de constater que Java EE offre, dans sa plateforme de service standard, une multitude de possibilité techniques.
 - JDBC : Support des SGBDR
 - JMS : Support de la messagerie applicative asynchrone
 - JCA : Support de connectivité d'applications diverses (ERP, CRM, Mainframe...) par des connecteurs dédiés
 - JavaMail : Support de la messagerie électronique
 - JNDI : Support des services d'annuaires
 - JAX-WS : Support des Services Web SOAP
 - RMI/IIOP : Support de l'invocation de méthodes distantes, compatible CORBA

Support de ces technologies dans Spring Framework

- Spring Framework met en avant les technologies les plus communément employées dans Java EE dans ses différents modules.
 - Spring DAO et Spring JDBC pour l'intégration des SGBDR par exemple.
- Le framework offre pour ces technologies d'intégration, un support avancé permettant de les mettre en œuvre en s'affranchissant de la complexité de la plateforme Java EE au travers d'API de haut niveau
- Ainsi, l'utilisation des Services Web, de la messagerie applicative JMS, de l'invocation de méthodes distantes, sont autant de mécanismes pris en charge par Spring Framework.
- Un autre aspect important de l'intégration technologique et la mise en œuvre d'une sécurité d'entreprise. Souvent mal comprise et clairement complexe dans Java EE, Spring facilite la mise en place d'une sécurité applicative uniforme.

La messagerie asynchrone Java

- Dans une communication classique entre objets logiciels, les échanges de données se font par appels de méthodes (localement ou à distance).
- La principale problématique de ce mode de fonctionnement est lié au fait que l'objet invoquant le traitement doivent en attendre la fin avant de pouvoir continuer son exécution.
 - C'est le cas typique d'une invocation « synchrone », donc, bloquante.
- Dans le contexte des applications Web notamment, une attente trop longue d'un résultat de traitement peut se solder par l'expiration d'un « timeout » et l'envoi d'un message d'erreur au client final.
 - Or, tous les traitements ne nécessite pas l'attente d'un résultat !
- La messagerie asynchrone Java (JMS : Java Message Service) résout ce problème en permettant l'envoi de messages applicatifs entre composants logiciels.
- Il n'est plus nécessaire qu'un émetteur de message attende le traitement de celui-ci pour pouvoir continuer son exécution.
 - Le facteur attend-t-il que vous ayez ouvert votre boîte au lettres avant de remonter dans sa camionnette ??
- Un intermédiaire (la boîte au lettres ...) va permettre d'éviter cette attente.

Principes de la messagerie asynchrone JMS

- Les acteurs d'une infrastructure JMS :
 - JMS Provider / JMS Server
 - Système de messagerie asynchrone, implémentant la spécification JMS, responsable du stockage et du routage des messages
 - Clients
 - Émetteurs ou récepteurs de messages
 - Messages
 - Données ou événements échangés entre les clients par l'intermédiaire du JMS Provider
 - Destinations
 - Emplacement de stockage des messages dans le JMS Provider. Elles peuvent être soit **File d'attente**, soit **Sujet**
- L'infrastructure JMS :
 - Pour pouvoir échanger des messages, les clients doivent :
 - Obtenir une connexion au JMS Server
 - Cette connexion est obtenue par une fabrique de connexions installée dans le serveur Java EE, et enregistrée dans le service de nommage JNDI
 - Obtenir une destination JMS
 - Les destinations sont créées et configurées par l'administrateur de serveur Java EE

Principes de la messagerie asynchrone JMS (suite)

- Domaines de messagerie asynchrone

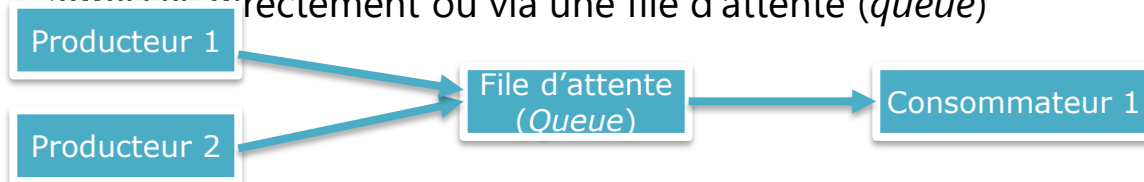
- Publication/Abonnement

- Les abonnés manifestent leur intérêt pour un sujet (*topic*) ; les éditeurs créent des messages qui sont distribués à tous les abonnés.

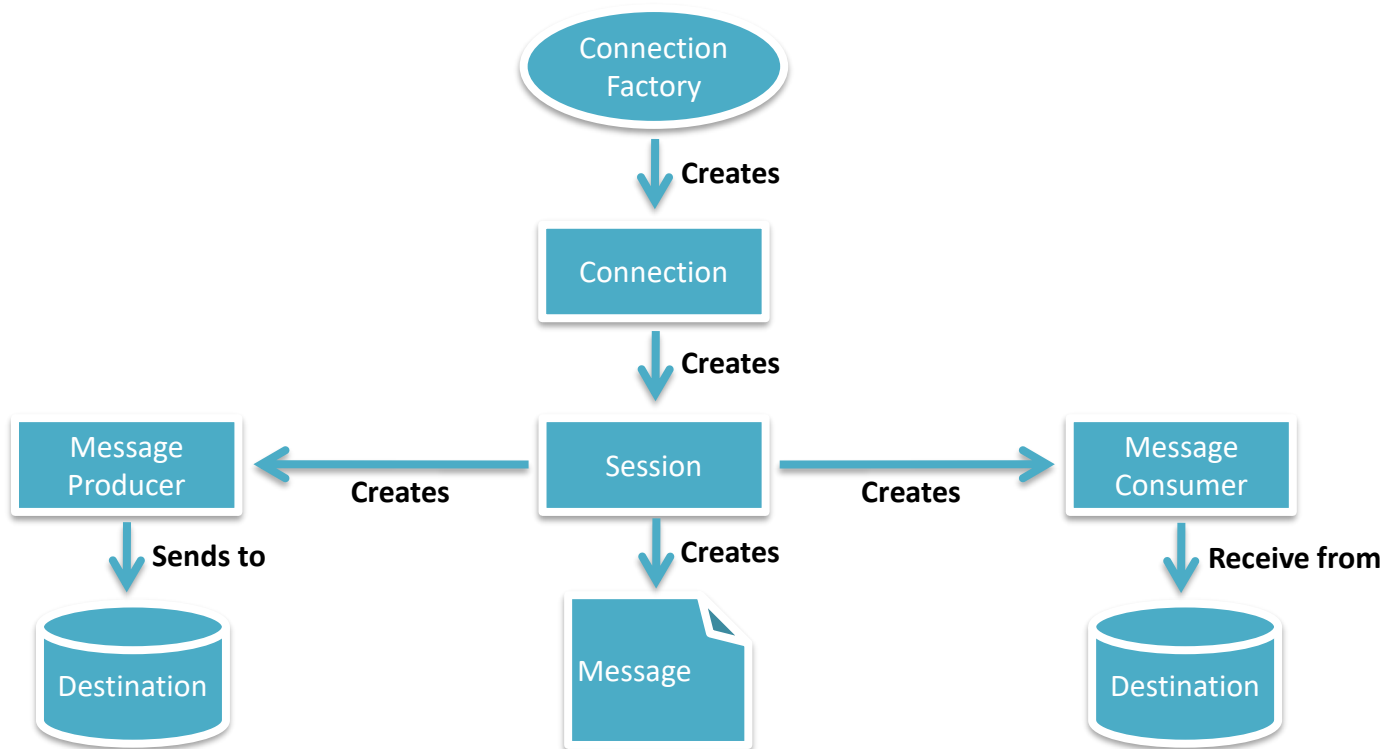


- Point à point

- Chaque message ne peut avoir qu'un consommateur ; les messages sont adressés directement ou via une file d'attente (*queue*)



Le modèle de programmation JMS



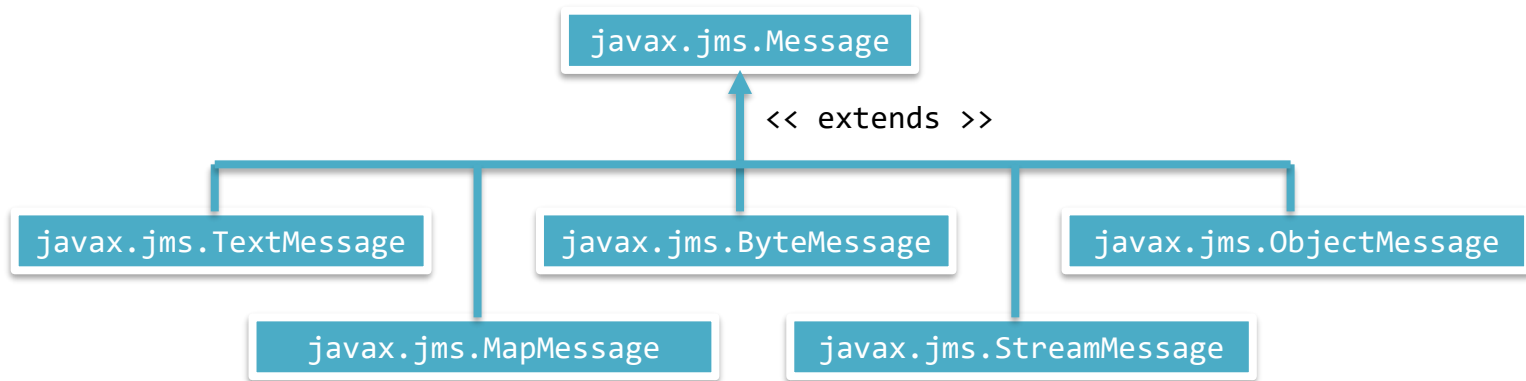
L'API JMS

- L'API JMS est située dans le packages `javax.jms`
- Les classes de l'API sont adaptées au domaine de messagerie applicative utilisé.

Interface Parent	Point à point	Publication/abonnement
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

Les messages JMS

- Il existe différents types de messages applicatifs en fonction du type d'information qu'ils sont dédiés à transporter.
- L'interface de base est `javax.jms.Message`



Utilisation de JMS avec Spring Framework

- Spring Framework permet de faciliter l'utilisation de la messagerie asynchrone JMS
- Par contre, il ne fournit pas d'implémentation de serveur JMS ! Il s'appuie sur l'implémentation par défaut du serveur d'application hébergeant l'application Java EE, ou sur une implémentation externe à ce serveur.
 - Les utilisateurs de serveurs d'applications Apache Tomcat pourront, par exemple s'appuyer sur le serveur ActiveMQ
 - ActiveMQ est un fournisseur JMS Open Source particulièrement léger et performant. Entièrement écrit en Java. Ce projet est maintenant un sous-projet du projet Geronimo d'Apache. <http://activemq.apache.org/>
- Spring Framework utilise l'espace de nommage jee dans sa configuration pour utiliser les services JNDI de localisation de service (locaux)
- Dans le cas de services JMS distant, il faudra les référencer sous forme de beans Spring

Configuration JMS avec Spring

- La classe de configuration doit être annotée avec `@EnableJms`

```
@Configuration
@ComponentScan({
    "fr.formation.banque.jms"
})
@EnableJms
public class ConfigurationJms {

}
```

Configuration de fabriques distantes

- Dans le cas de serveurs JMS externes, la configuration se fait sous forme de bean Spring et sans localisation JNDI.
- Il faut évidemment que les JARs permettant la connexion distante soient disponibles dans le classpath de l'application

```
@Bean("jmsConnectionFactory")
public ActiveMQConnectionFactory jmsConnectionFactory() {
    ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory();
    factory.setBrokerURL("tcp://localhost:61616?wireFormat.maxInactivityDuration=0");

    List<String> trustedPackages = new ArrayList<>();
    trustedPackages.add("fr.formation.banque.jms");
    factory.setTrustedPackages(trustedPackages);

    return factory;
}
```

Localisation des destinations JMS

- Une fois la fabrique de connexions localisée, il faut localiser la destination JMS utilisé par le producteur et le consommateur de message.

```
@Bean("destination")
public ActiveMQQueue destination() {
    return new ActiveMQQueue("banqueQueue");
}
```

Le JmsTemplate

- Le template JMS est la classe centrale du support JMS de Spring puisqu'elle facilite l'interaction entre le fournisseur JMS et l'application.
- Ce template s'appuie sur une fabrique de connexions JMS et une destination configurées de la même manière que précédemment.
 - La fabrique et la destination seront injectés dans le template
- Il faut ensuite injecter ce template dans les classes d'implémentation qui vont envoyer les messages JMS

```
@Bean("jmsTemplate")
public JmsTemplate jmsTemplate() {
    JmsTemplate jmsTemplate = new JmsTemplate();
    jmsTemplate.setConnectionFactory(jmsConnectionFactory());
    jmsTemplate.setDefaultDestination(destination());
    return jmsTemplate;
}
```

Exemple d'envoi de message

```
public class JmsProducer {  
    private JmsTemplate jmsTemplate;  
  
    public void setJmsTemplate(final JmsTemplate jmsTemplate) {  
        this.jmsTemplate = jmsTemplate;  
    }  
  
    public void envoyerMessage() {  
        jmsTemplate.send(new MessageCreator() {  
            public Message createMessage(final Session session)  
                throws JMSEException {  
                return session.createTextMessage("Message " + new Date());  
            }  
        });  
    }  
}
```

Réception de messages

- La réception de messages doit se faire de manière asynchrone.
- Spring met à disposition la classe `DefaultMessageListenerContainer` pour cela.
- Cette implémentation va permettre d'écouter des messages JMS entrant en utilisant l'interface `javax.jms.MessageListener`
 - Il est nécessaire de déclarer l'espace de nommage `jms` dans la configuration Spring

```
@Bean("jmsListenerContainerFactory")
public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() {
    DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();
    factory.setConnectionFactory(jmsConnectionFactory());
    return factory;
}
```


L'implémentation du récepteur

```
@JmsListener(destination = "banqueQueue", containerFactory = "jmsListenerContainerFactory")
public void recevoirMessage(Message message) {
    try {
        // ...
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
```

Travaux Pratiques





Module 8

La sécurité des applications Spring

Contenu du module

- Rappels sur la sécurité des applications Java avec JAAS
- La sécurité dans Spring avec Spring Security (filtres de sécurité/URLs/annotations)
- Configuration de l'authentification et des autorisations d'accès
- La sécurité appliquée à l'invocation des beans
- Implémenter Spring Security dans les JSP

Sécurité des applications Java / Java EE

- Java propose deux API pour gérer la sécurité :
 - JAAS, pour les projets Java standards (J2SE),
 - et une implémentation spécifique de JAAS incluse dans la norme Java EE.
- JAAS (Java Authentication and Authorization Service)
 - intégré à J2SE 1.4 (optionnel auparavant).
 - API de bas niveau
 - La gestion des utilisateurs en base de données ou la création de formulaires Web de login ne sont pas couverts par cette spécification.
 - Mais une API permettant de développer des modules de connexion est fournie
- Dans Java EE
 - Plusieurs objets et méthodes dédiés à la sécurité.
 - Intégrés dans les frameworks existants.
 - Standard officiel.

La sécurité Java EE

- Sécurisation des modules Web
 - La spécification Servlet permet de définir des règles de sécurité au niveau du fichier `web.xml` (ou par annotations) afin de protéger des URL en fonction de leur nom.
 - Elle supporte plusieurs méthodes d'authentification simples (par formulaire, authentification HTTP basique ou certificat) et fournit une API basique.
- Sécurisation des modules EJB
 - La spécification EJB permet de déclarer des restrictions d'invocations sur les méthodes de chaque composants.
 - Cette déclaration se fait par annotations ou bien via le fichier `ejb-jar.xml`

Inconvénients du modèle de sécurité Java EE

- Plus ou moins portable d'un serveur d'applications à un autre
- La gestion des URL dans le fichier **web.xml** est rudimentaire.
 - Impossible par exemple, d'utiliser des expressions régulières pour définir une URL.
- Services fournis sont généralement élémentaires.
 - Pas d'authentification automatique par cookie,
 - Pas de système pour empêcher deux utilisateurs d'utiliser le même login en même temps,
 - etc.
- Verbose des déclarations dans les descripteurs de déploiement ou dans les annotations
- Manque d'homogénéité entre les syntaxes Web et EJB pour les déclarations

Spring Security

- Spring Security a pour objectif de proposer un système complet de gestion de la sécurité.
- Spring Security est un module additionnel au framework
 - <http://projects.spring.io/spring-security>
- Avantages de Spring Security
 - Portabilité : il ne dépend pas d'un serveur d'applications particulier.
 - Il fournit en standard un nombre de fonctionnalités beaucoup plus important qu'un serveur Java EE classique
 - l'authentification automatique par cookie, la vérification qu'un utilisateur n'est pas déjà authentifié, ...
 - Support de solutions de Single Sign-On (SSO)
 - Gestion des utilisateurs : solution intégrée, complète et portable.
 - Sécurisation des requêtes HTTP : disponible de manière plus fine que dans la norme Java EE.
 - Sécurisation de la couche de service : API complète, qui s'intègre dans les frameworks courants d'AOP.
 - Sécurisation de la couche de domaine : listes de contrôle d'accès (ACL), qui peuvent être spécifiées au niveau de chaque objet de domaine.

Installation de Spring Security

- La distribution Spring Security contient un certain nombre de fichiers JAR qu'ils faut intégrer au projet en fonction des besoins
 - `spring-security-core`, `spring-security-config`, `spring-security-web`, ...
- La configuration interviendra à plusieurs niveaux selon le type de ressources à sécuriser
 - Obligation d'une classe de configuration Spring
 - Annotée avec `@Configuration`
 - La sécurité des ressources Web nécessitera :
 - La déclaration d'un filtre de servlet dans le descripteur de déploiement de l'application
 - L'ajout de l'annotation `@EnableWebSecurity` sur la classe de configuration
 - L'héritage de `WebSecurityConfigurerAdapter` sur la classe de configuration

Configuration Web de Spring Security

- Déclaration d'un filtre de Servlet dans le fichier web.xml

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- Le nom de filtre `springSecurityFilterChain` est important à respecter car il fait référence à l'id d'un bean Spring Security
- L'URL pattern spécifie l'application du filtre sur les requêtes
- Configuration Web :

```
@Configuration
@EnableWebSecurity
public class ConfigurationSecurite extends WebSecurityConfigurerAdapter {
}
```

Gestion de l'authentification

- Le composant effectuant l'authentification dans Spring Security est un `AuthenticationManager`.
 - Le framework fournit plusieurs implémentations par défaut.
- La configuration permet ensuite de spécifier l'objet `AuthenticationManager` à utiliser via la méthode `configureGlobal()`
 - Le paramètre est de type `AuthenticationManagerBuilder`
 - Elle doit être annotée avec `@Autowired`
- Mise en place dans la configuration :

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    // Implémentation du registre d'utilisateurs...
}
```

Gestion de l'authentification (suite)

- On peut ensuite fournir la configuration du registre d'utilisateur à exploiter selon son type
 - En mémoire
 - `auth.inMemoryAuthentication()`
 - Définition de comptes utilisateurs directement dans le code (en développement ou pour les tests)
 - JDBC
 - `auth.jdbcAuthentication()`
 - LDAP
 - `auth.ldapAuthentication()`
- L'objet retourné par ces différentes méthode devra être configuré en spécifiant ses propriétés

Gestion de l'authentification (suite)

- Authentification en mémoire
 - Idéal en développement / test
 - On spécifie :
 - Un nom d'utilisateur
 - `withUser()`
 - Un mot de passe
 - `password()`
 - Un rôle applicatif
 - `authorities()`

```
auth.inMemoryAuthentication()  
    .withUser("user")  
    .password("password")  
    .authorities("USER")  
;
```

Gestion de l'authentification (suite)

- Authentification dans une base de données

```
auth.jdbcAuthentication()  
    .dataSource(dataSource()) ← Source de données pour la connexion à la base  
    .usersByUsernameQuery( ← Sélection des informations utilisateur  
        "select id,password from app_users where id=?"  
    )  
    .authoritiesByUsernameQuery( ← Sélection des rôles utilisateur  
        "select id,role from app_roles where id=?"  
    )  
;
```

Encodage des mots de passe

- Il convient généralement de chiffrer les mots de passe dans les différents référentiels de stockage
- Spring Security permet de spécifier un bean de type PasswordEncoder pour appliquer un algorithme de chiffrement aux mots de passe

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("user")
        .password(passwordEncoder().encode("password"))
        .authorities("USER");
}
```

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Une implémentation de PasswordEncoder

Sécurité d'une application Web

- La sécurité des applications Web se base sur le filtrage des URLs grâce au filtre de Servlet positionné par Spring Security dans la configuration principale de l'application (web.xml).
- Le filtrage des URL se définit dans la méthode configure(HttpSecurity http) de la classe de configuration

```
@Configuration
@EnableWebSecurity
public class ConfigurationSecurite extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        // ...
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // ...
    }
}
```


Configuration des restrictions

- Les restrictions d'accès sur les URLs se configurent grâce aux méthodes de l'objet `HttpSecurity`
- Première étape :
 - Ecrire une expression qui permet d'indiquer les URLs concernées
 - `.antMatcher("/admin/*")`
 - Ou préciser que toutes les URLs sont concernées...
 - `.anyRequest()`
- Seconde étape :
 - Indiquer le(s) rôle(s) nécessaire(s) pour pouvoir accéder à la ressource
 - `.hasAuthority("ADMIN")`
 - `.hasAnyAuthority("ADMIN", "USER")`
 - Où radicalement...
 - `.permitAll()`
 - `.denyAll()`

Mécanismes d'authentification

- Le schéma d'authentification doit enfin être spécifié
- Avec Spring Security on utilise généralement :
 - L'authentification standard HTTP Basic
 - L'authentification par formulaire
- Le schéma d'authentification se précise via l'objet `HttpSecurity`
 - `.httpBasic()`
 - `.formLogin()`
 - Il sera nécessaire de préciser :
 - L'URL de la page de connexion
 - L'URL du contrôleur vers lequel envoyer les données d'authentification
 - L'URL de redirection en cas de succès
 - L'URL de redirection en cas d'échec

Authentication : Exemples complets

- Authentication HTTP

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/*")
        .hasAuthority("ADMIN")
        .and()
        .httpBasic()
    ;
}
```

- Authentication par formulaire

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/*")
        .hasAuthority("ADMIN")
        .and()
        .formLogin()
        .loginPage("/login.html")
        .loginProcessingUrl("/perform_login")
        .defaultSuccessUrl("/homepage.html", true)
        .failureUrl("/login.html?error=true")
    ;
}
```



Travaux Pratiques



Fin de la formation

Spring Framework : Mise en œuvre

Pour aller plus loin

- ENI Service sur Internet
 - Consultez notre site web www.eni-service.fr
 - Les actualités
 - Les plans de cours de notre catalogue
 - Les filières thématiques et certifications
 - Abonnez-vous à nos newsletters pour rester informé sur nos nouvelles formations et nos événements en fonction de vos centres d'intérêts.
 - Suivez-nous sur les réseaux sociaux
 -  Twitter : <http://twitter.com/eniservice>
 -  Viadeo : <http://bit.ly/eni-service-viadeo>

Pour aller plus loin

- Notre accompagnement
 - Tous nos Formateurs sont également Consultants et peuvent :
 - Vous accompagner à l'issue d'une formation sur le démarrage d'un projet.
 - Réaliser un audit de votre système d'information.
 - Vous conseiller, lors de vos phases de réflexion, de migration informatique.
 - Vous guider dans votre veille technologique.
 - Vous assister dans l'intégration d'un logiciel.
 - Réaliser complètement ou partiellement vos projets en assurant un transfert de compétence.

Votre avis nous intéresse

Nous espérons que vous êtes satisfait de votre formation.

Merci de prendre quelques instants pour nous faire un retour en remplissant le questionnaire de satisfaction.

Merci pour votre attention,
et au plaisir de vous revoir prochainement.