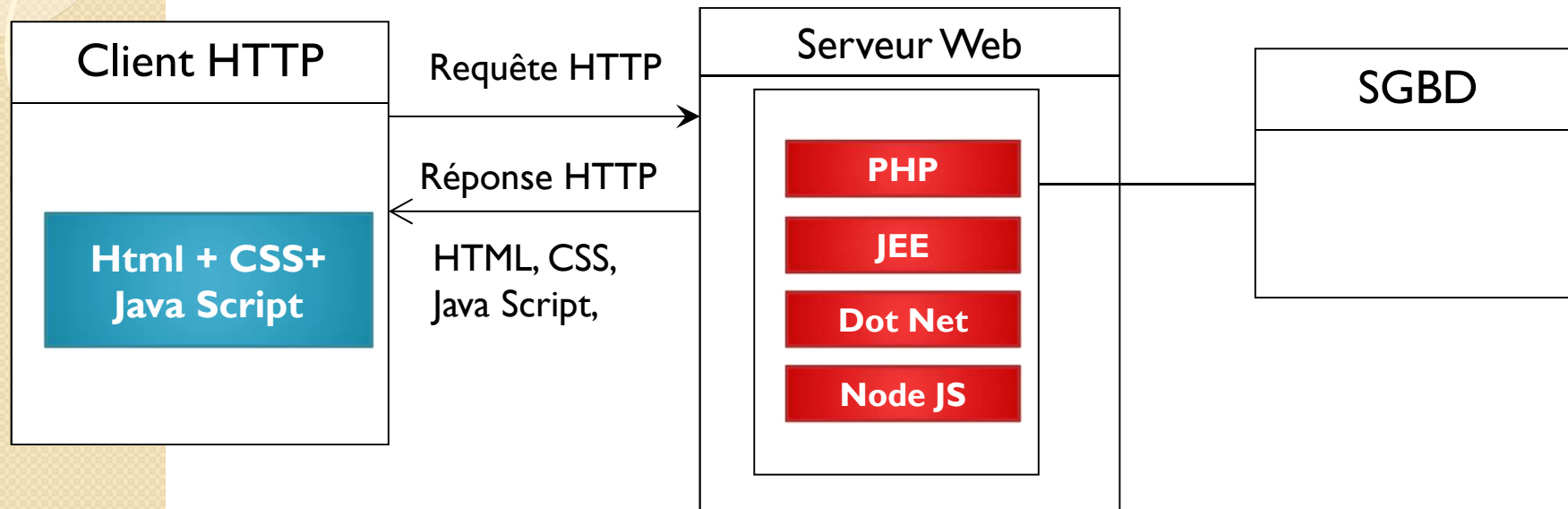




# Architecture JEE

## Principe de l'inversion de contrôle et d'injection des dépendances

# Architecture Web



- Un client web (Browser) communique avec le serveur web (Apache) en utilisant le protocole HTTP
- Une application web se compose de deux parties:
  - La partie **Backend** : S'occupe des traitements effectués coté serveur :
    - Technologies utilisées : PHP, JEE, .Net, Node JS
  - La partie **Frontend** : S'occupe de la présentations des IHM coté Client :
    - Langages utilisés : HTML, CSS, Java Script
- La communication entre la partie Frontend et la partie backend se fait en utilisant le protocole HTTP

# Exigences d'un projet informatique

## Exigences Fonctionnelles

Satisfaire les besoins fonctionnels (métiers) de l'entreprise

## Exigences Techniques

Performances : l'application doit être performante

Temps de réponse

Problème de montée en charge : Vers une architecture Distribuée scalable

Equilibrage de charges et Tolérances aux pannes

Maintenance : l'application doit être facile à maintenir

Une application doit évoluer dans le temps

L'application doit être fermée à la modification et ouverte à l'extension

Sécurité : L'application doit prévoir des solutions pour toutes les failles de sécurité

Persistances des données dans des SGBD appropriés, Gestion des Transactions

Créer différents types de clients pour accéder à l'application : Web, Mobile, Desktop,

.....

Exigences financières : Le coût du logiciel doit respecter les contraintes budgétaires

# Constat

- Il est très difficile de développer un système logiciel qui respecte ces exigences sans **utiliser l'expérience des autres** :
  - Bâtir l'application sur une architecture d'entreprise:
    - JEE
  - **Framework pour l'Inversion de contrôle:**
    - **Permettre au développeur de s'occuper uniquement du code métier (Exigences fonctionnelles) et c'est le Framework qui s'occupe du code technique (Exigences Techniques)**
      - **Spring (Conteneur léger)**
      - **EJB (Conteneur lourd)**
  - Frameworks :
    - Mapping objet relationnel (ORM ) : JPA, Hibernate, Toplink, ...
    - Applications Web : Struts, JSF, SpringMVC
  - Middlewares :
    - RMI, CORBA : Applications distribuées
    - Web Services :
      - JAXWS pour Web services SOAP
      - JAXRS pour les Web services RESTful
    - Communication asynchrone entre les application :
      - JMS :
      - AMQP :

## Exemple : sans utiliser d'inversion de contrôle

```
public void virement(int c1, int c2, double mt) {
```

```
    /* Création d'une transaction */
```

```
    EntityTransaction transaction=entityManager.getTransaction();
```

```
    /* Démarrer la transaction */
```

```
    transaction.begin();
```

```
    try {
```

Code Technique

```
        /* Code métier */
```

```
        retirer(c1,mt);
```

```
        verser(c2,mt);
```

Code Métier

```
        /* Valider la transaction */
```

```
        transaction.commit();
```

```
    } catch (Exception e) {
```

```
        /* Annuler la transaction en cas d'exception */
```

```
        transaction.rollback();
```

```
        e.printStackTrace();
```

```
    }
```

```
}
```

Code Technique

## Exemple : en utilisant l'inversion de contrôle

**@Transactional**

```
public void virement(int c1, int c2, double mt) {  
    retirer(c1,mt);  
    verser(c2,mt);  
}
```

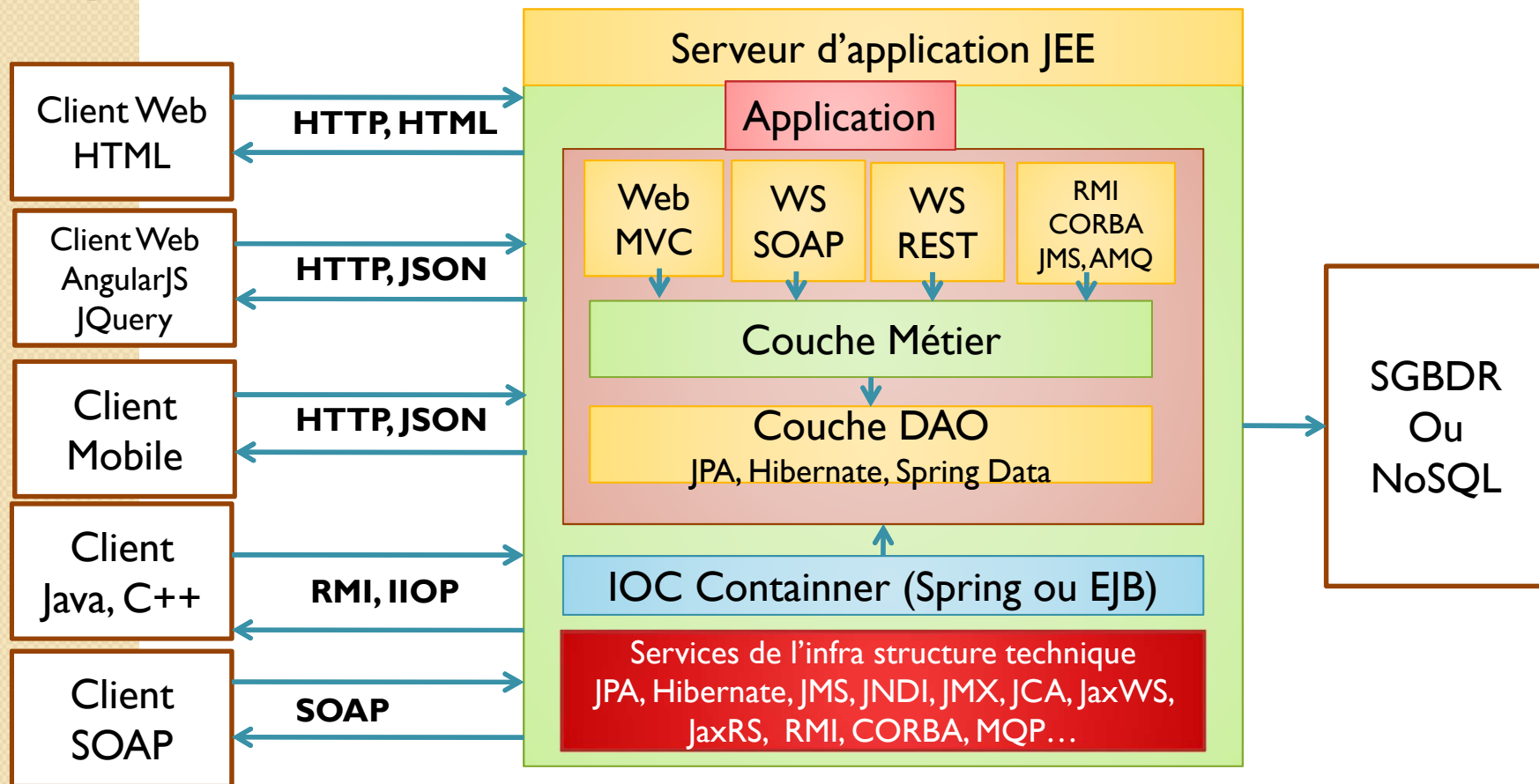
Code Métier

Ici, avec l'annotation **@Transactional**, nous avons délégué la gestion des transactions au conteneur Spring IOC

La séparation des aspects métiers et des aspects techniques d'une application est rendu possible grâce à la Programmation Orientée Aspect (AOP)  
En utilisant des tisseurs d'aspects comme :

- **AspectJ**
- **Spring AOP**
- **JBOSS AOP**

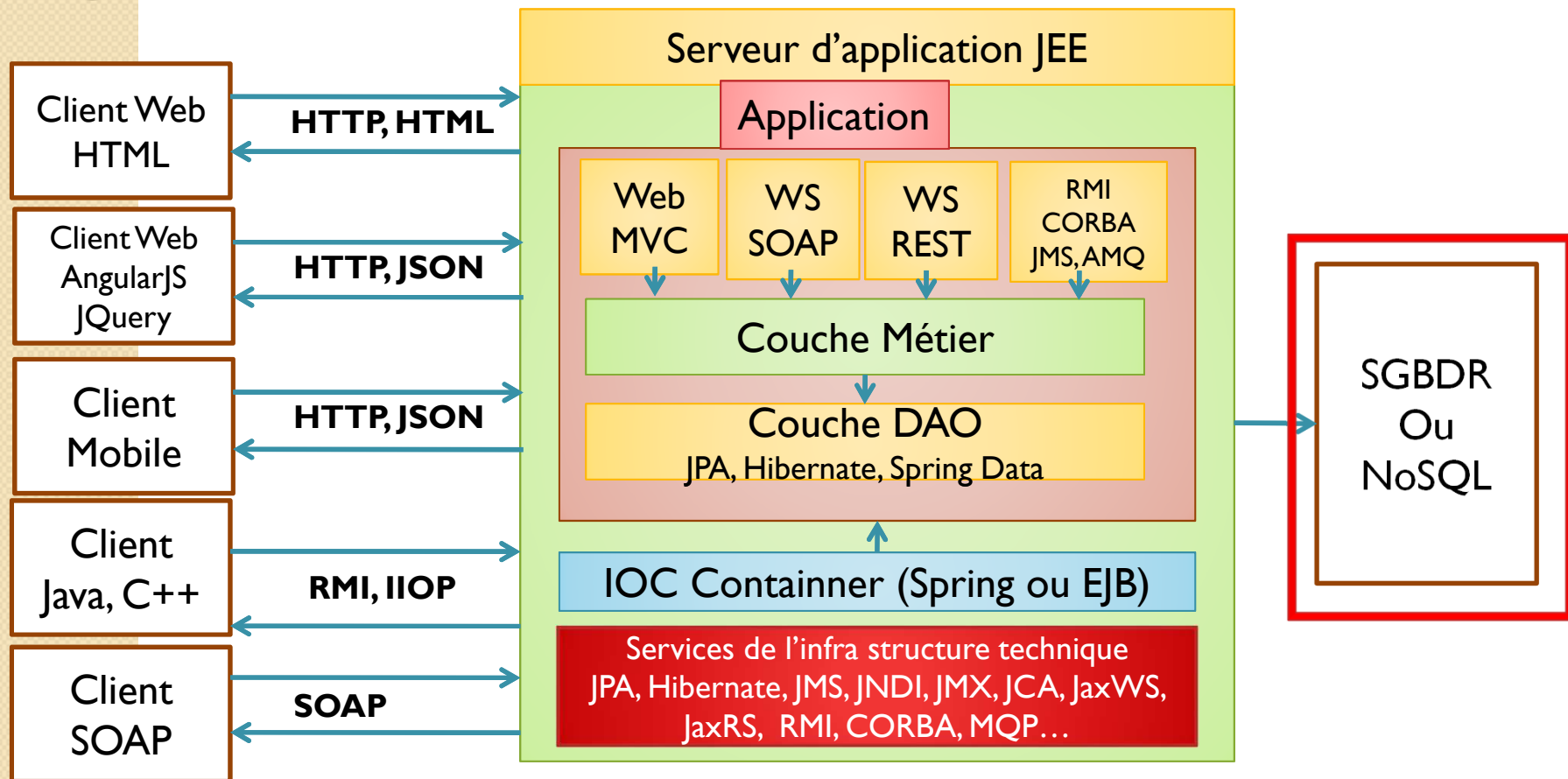
# Architecture d'une application



# Architecture d'une application

## SGBD :

- Permet de stocker les données de l'application
- Dans la majorité des cas il s'agit d'un SGBD relationnel (SqlServer, MySQL, Oracle, ...)
- Si la quantité de données sont très importante, on utilise des SGBD NoSQL (Not Only SQL) comme MangoDB, ...

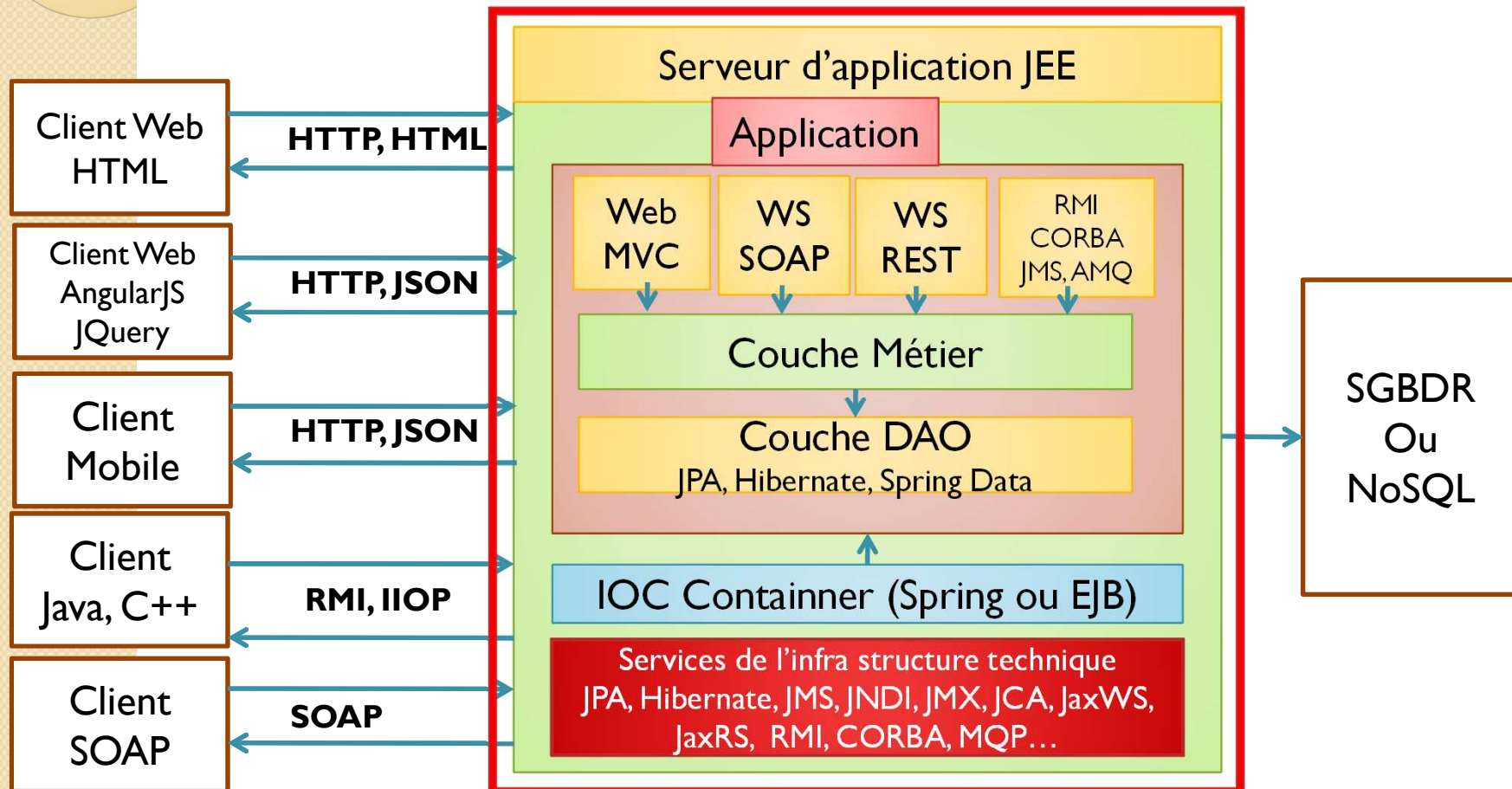




# Architecture d'une application

## Serveur d'application :

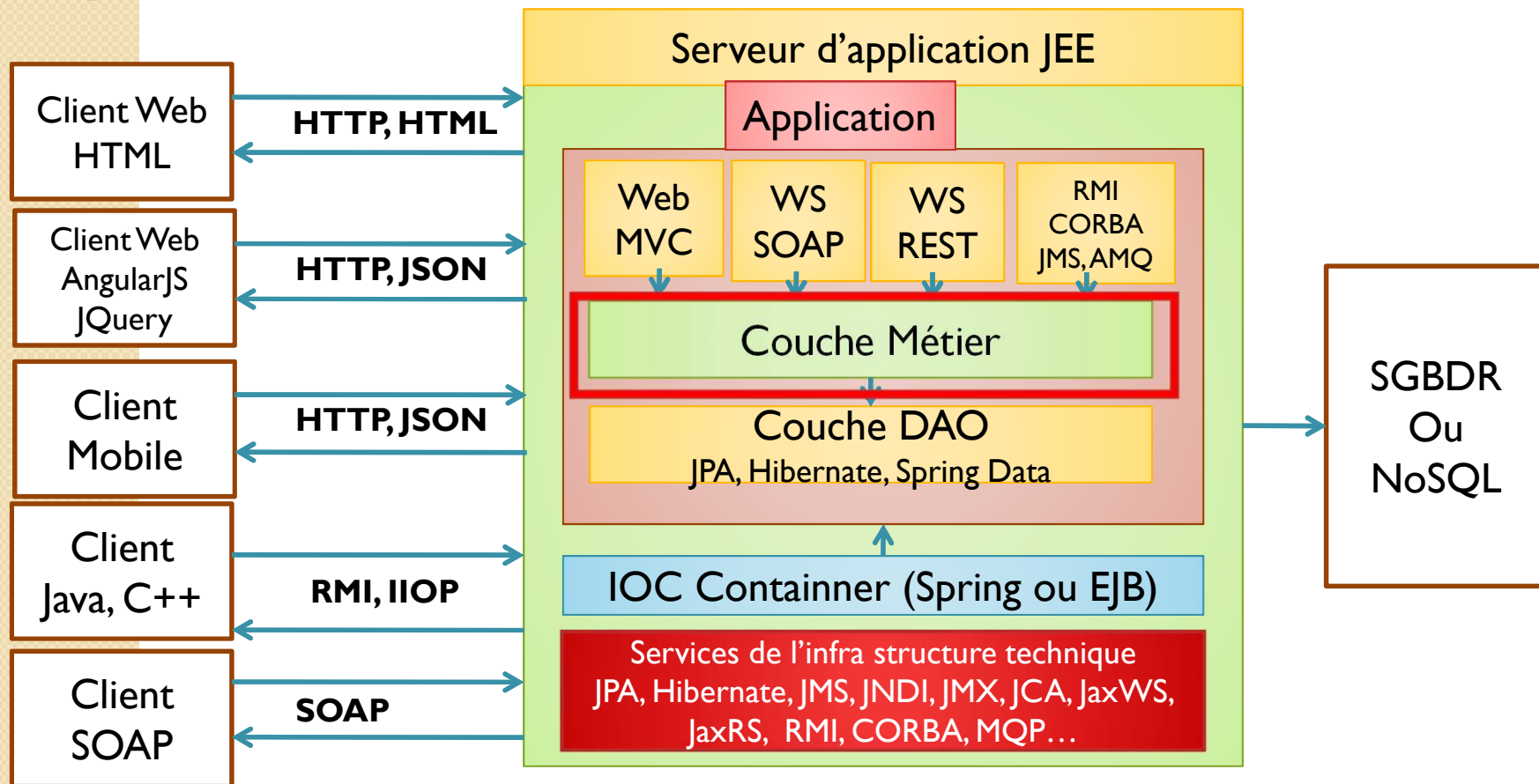
- Un serveur d'application qui permet de déployer les applications
- Il permet de gérer le cycle de vie des applications
- Fournie l'infrastructure nécessaire pour faire fonctionner les applications de bonnes qualités
- Offre un Framework d'Inversion de Contrôle (IOC) : Spring IOC ou EJB
- Exemple : JBoss, Glashfish, Web Sphere, ou encore (Tomcat+Spring,) etc..



# Architecture d'une application

## Couche Métier :

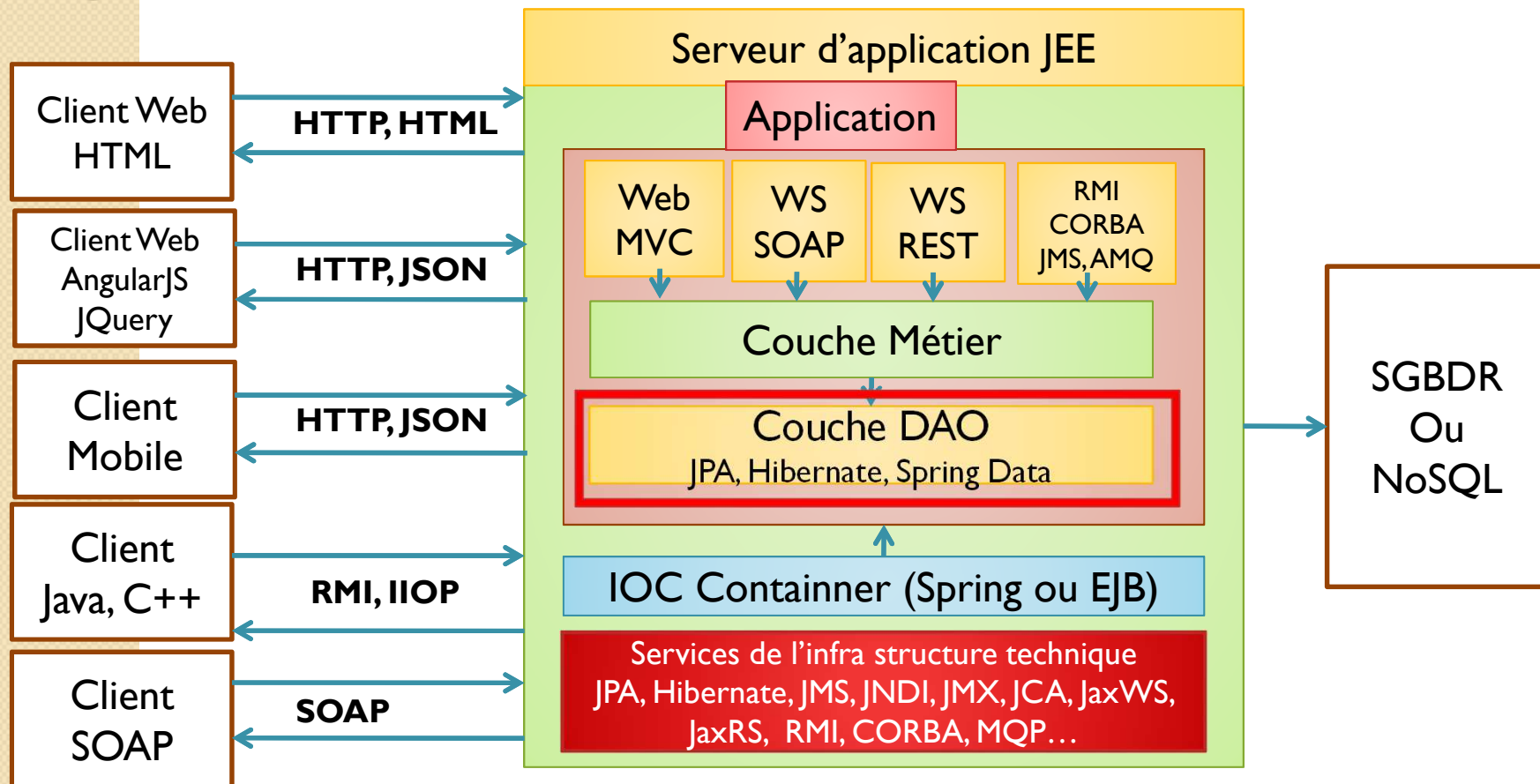
- Permet d'implémenter la logique métier de l'application
- Utilise généralement une approche orientée objet



# Architecture d'une application

## Couche DAO:

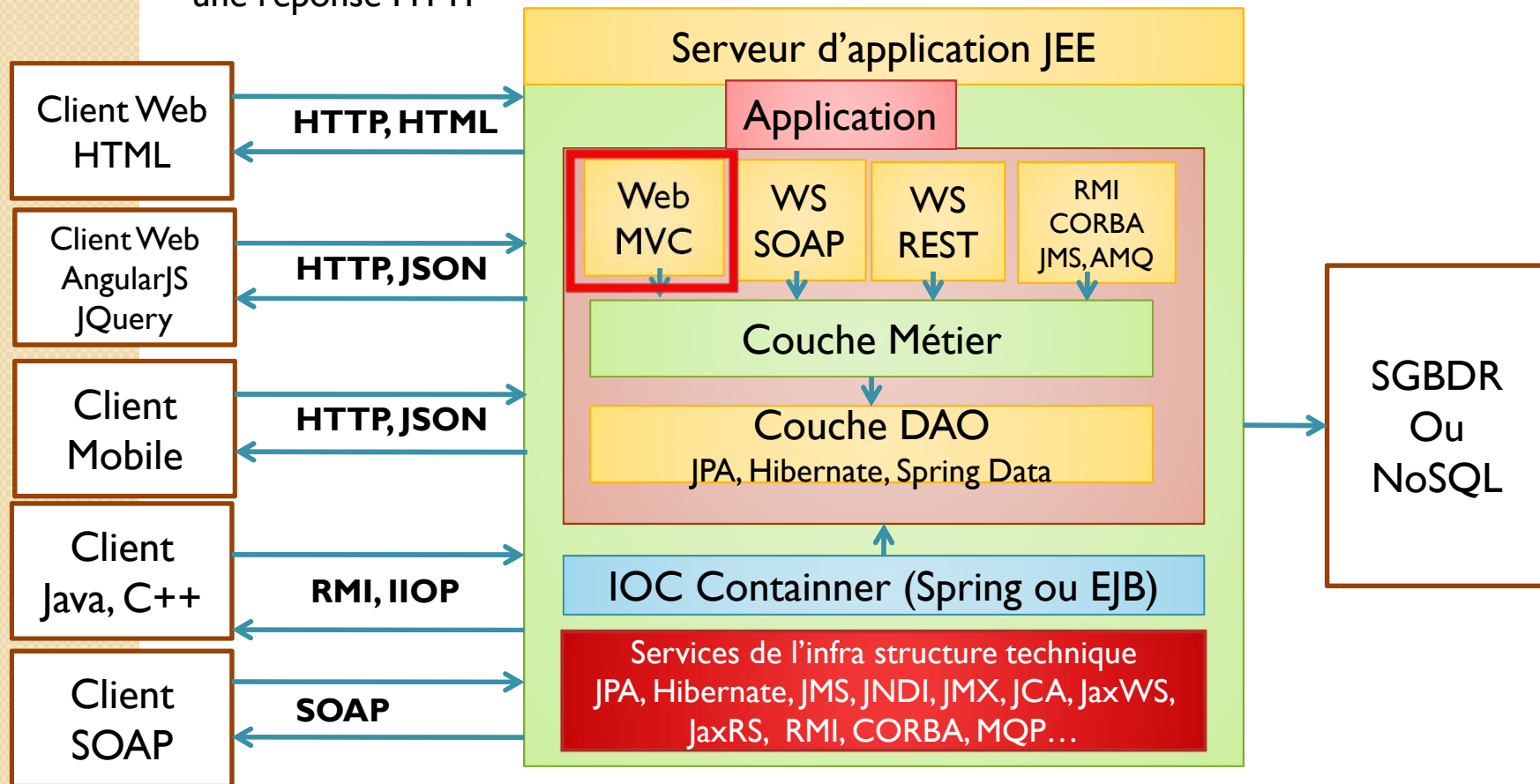
- Couche technique qui représente la couche d'accès aux données de l'application
- Si les données sont stockées dans une base de données relationnelle, cette couche utilise un Framework de Mapping Objet relationnel implémentant la spécification JPA comme Hibernate, TopLink, etc..



# Architecture d'une application

## Couche Web:

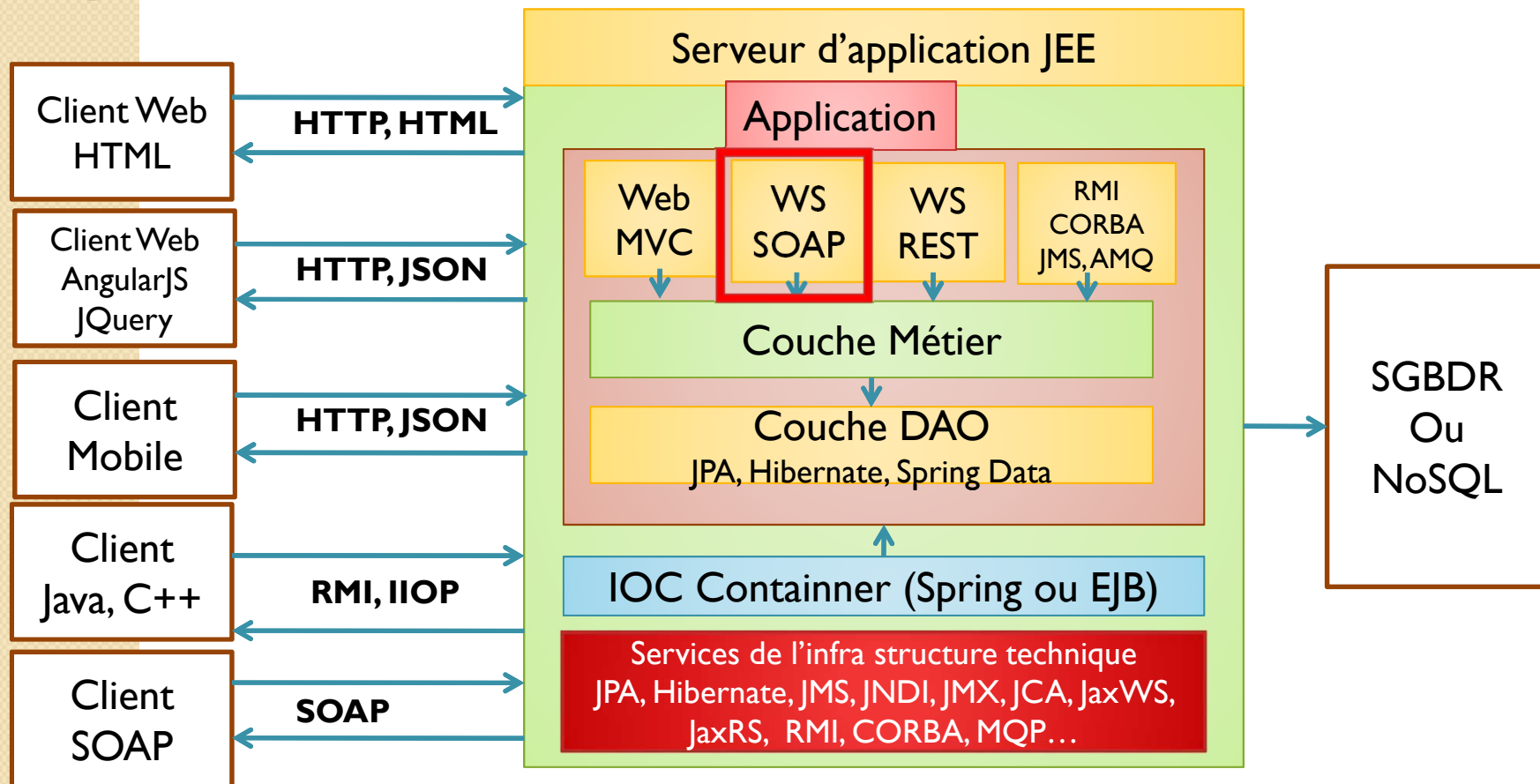
- Permet d'implémenter la logique présentation Web de l'application ( Servlet et JSP )
- Dans cette partie on utilise des Framework MVC comme Spring MVC, Struts, JSF, ..
- Cette couche reçoit les requêtes HTTP du client web
- Faire appel à la couche métier pour effectuer les traitements
- Envoie un résultat HTML au client, en utilisant un moteur de Template comme Thymeleaf dans une réponse HTTP



# Architecture d'une application

## Couche Web Services (SOAP):

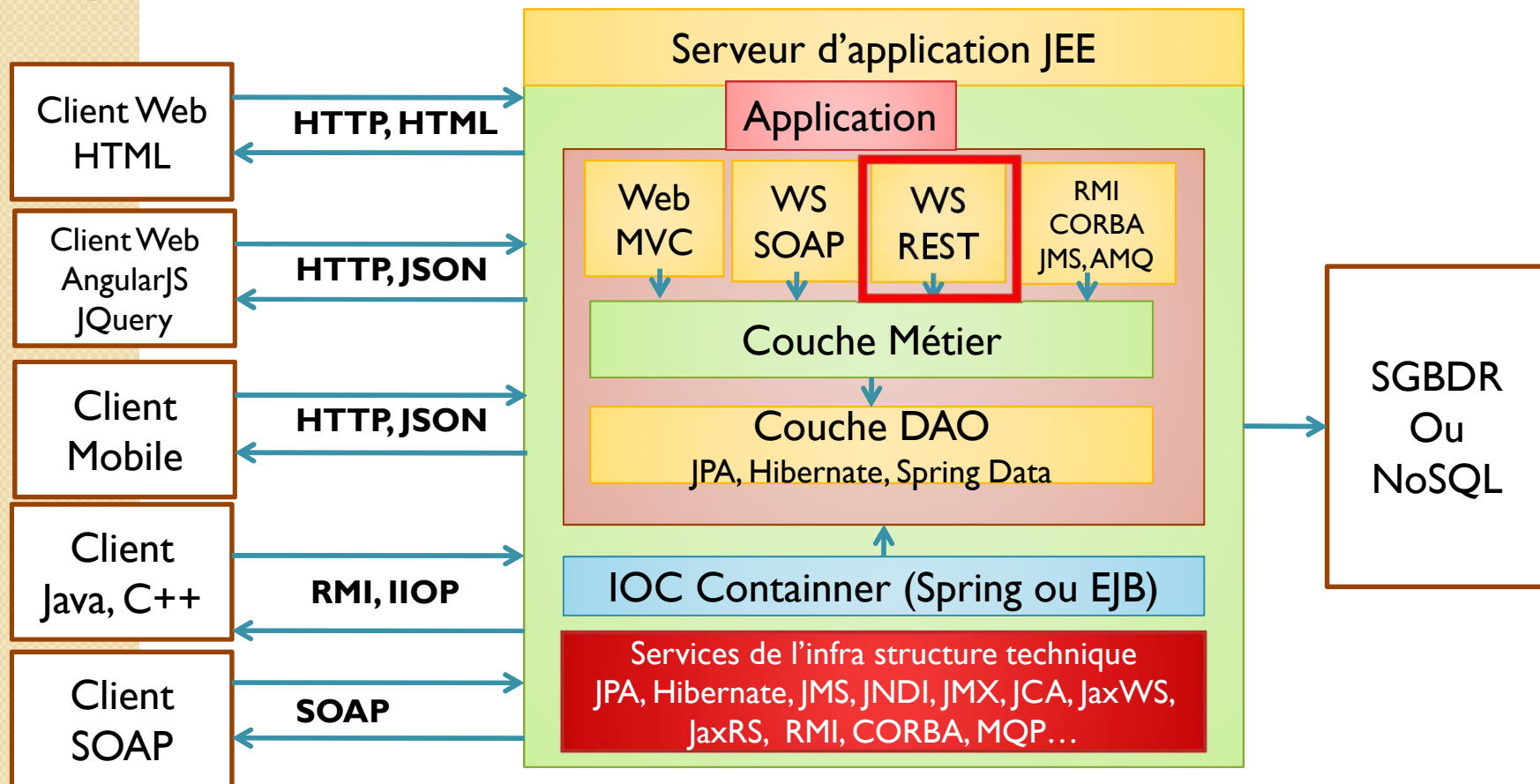
- Permet de définir un web service qui permet à d'autres applications développées avec d'autres langages de faire appel, à distance, aux fonctionnalités de l'application
- Le web service reçoit des requête HTTP contenant des messages XML (Requête SOAP)
- Exécute des traitement et renvoie le résultat au format XML (Réponse SOAP)
- Pour implémenter les Web service on utilise la spécification JAXWS



# Architecture d'une application

## Couche Web Services REST full:

- Permet de définir un web service qui permet à d'autres applications développées avec d'autres langages de faire appel, à distance, aux fonctionnalités de l'application
- Le service REST reçoit des requête HTTP
- Exécute des traitement et renvoie le résultat au client avec différents formats (JSON, XML)
- Pour implémenter les Web service REST on utilise la spécification JAXRS

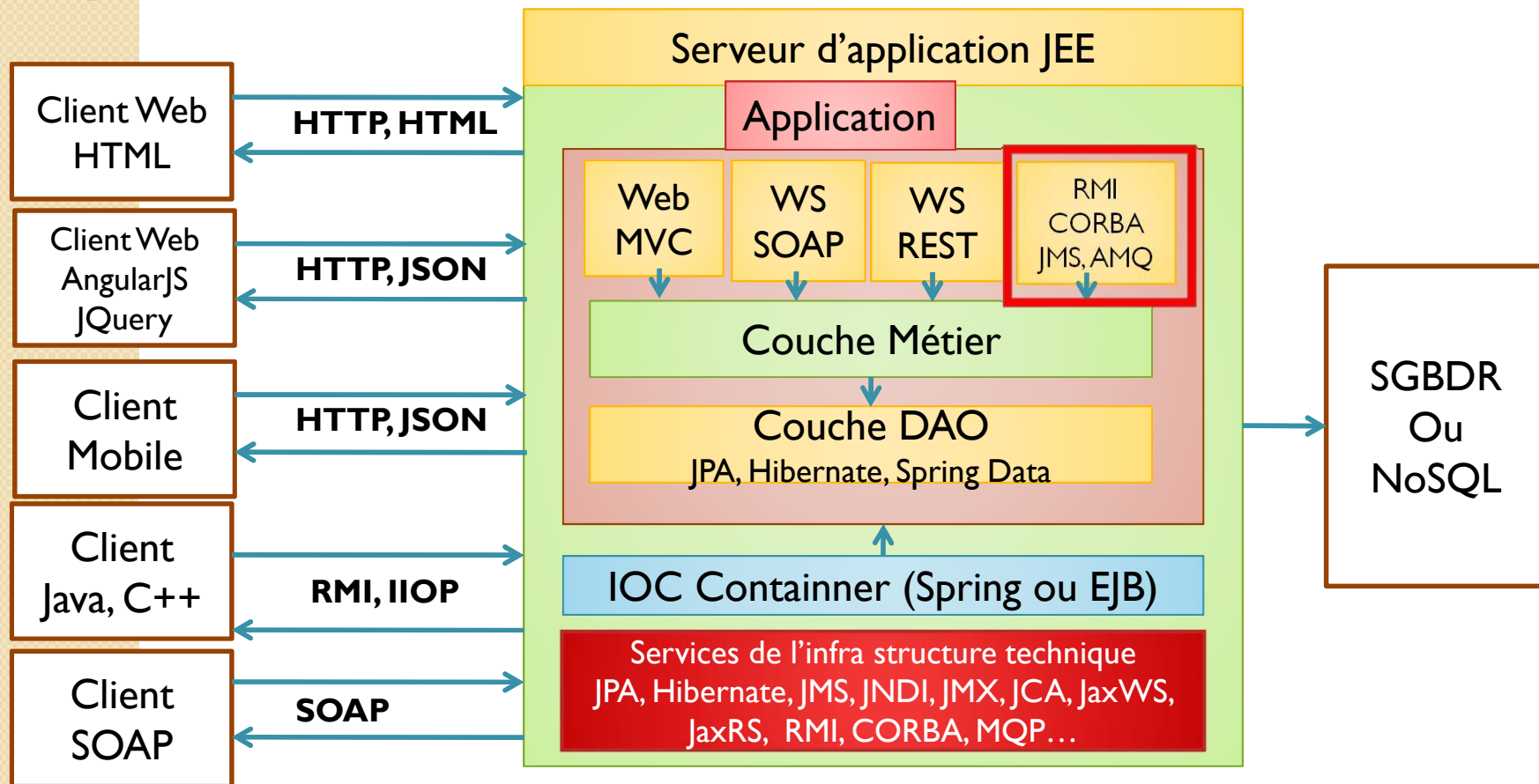




# Architecture d'une application

## Middlewares RMI, CORBA, JMS:

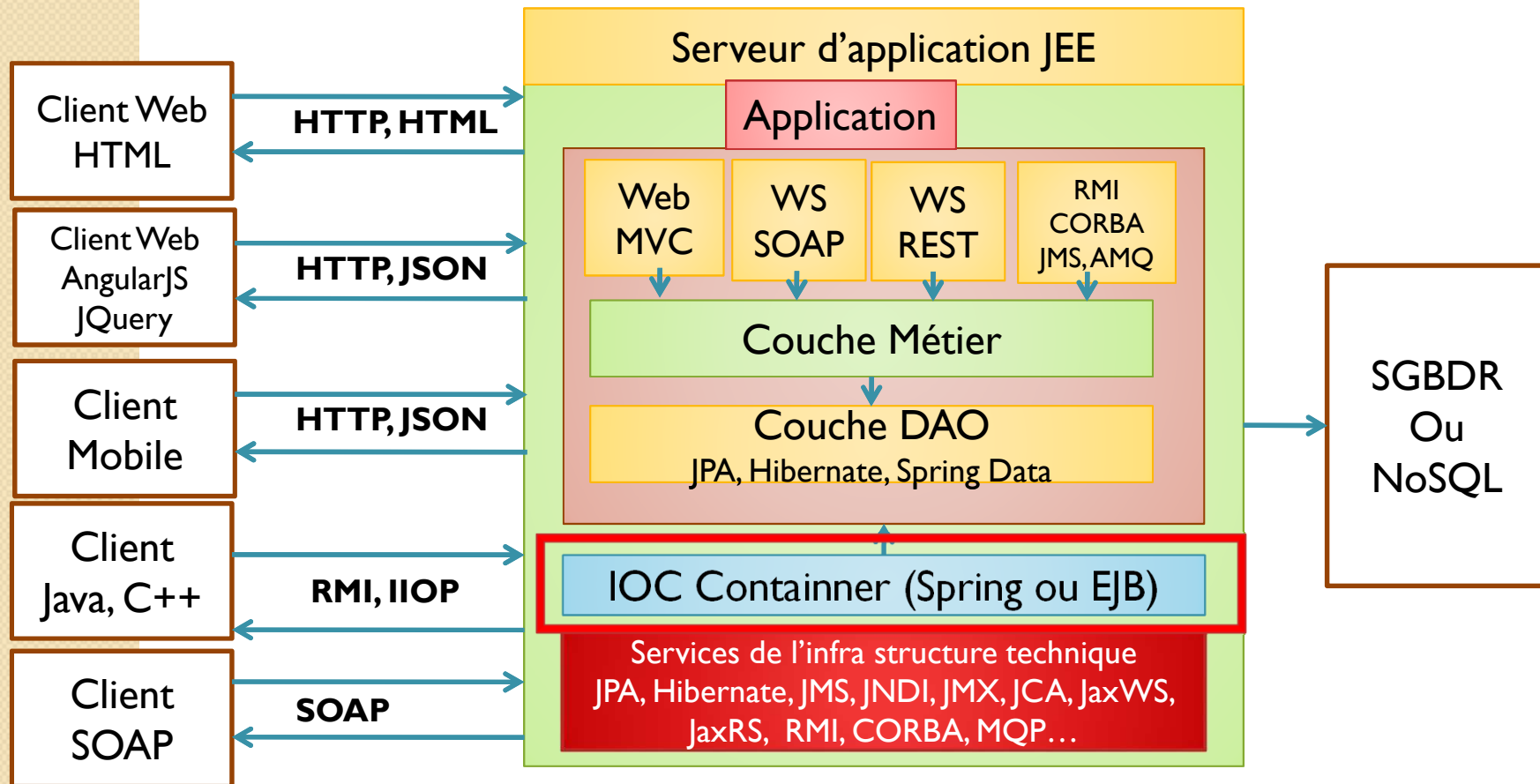
- RMI est utilisé pour créer des application distribuées Java
- CORBA est utilisé pour créer des applications distribuées hétérogènes
- JMS et AMQP sont utilisés pour les applications distribuées asynchrones



# Architecture d'une application

## IOC Container:

- Représente le Framework qui permet de faire l'inversion de contrôle
- Permet à l'application de se concentrer uniquement sur le code métier (Exigences fonctionnelles)
- Le Container IOC offre à l'applications les services techniques (Sécurité, Gestion de transaction, ORM, etc..)

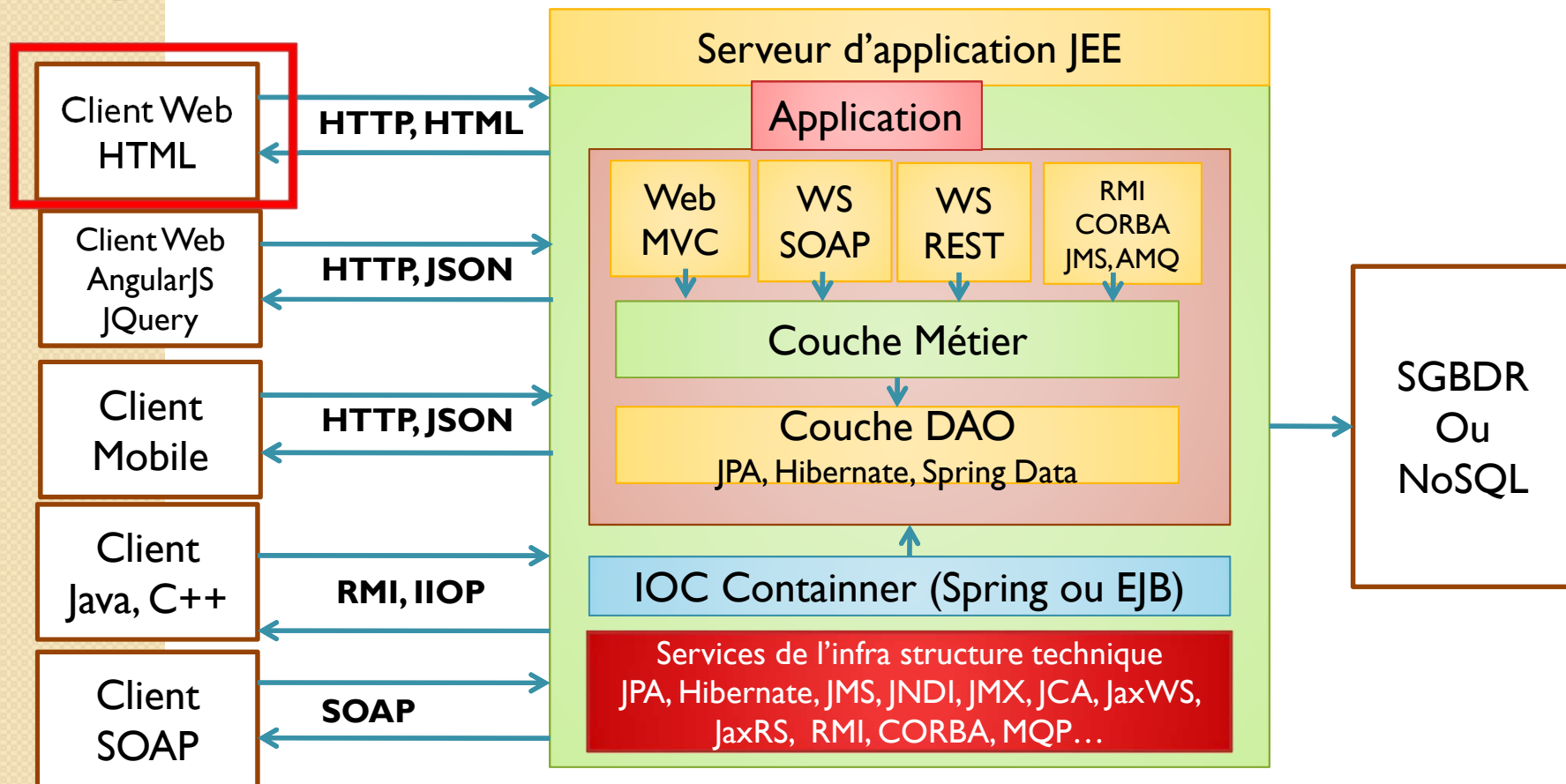




# Architecture d'une application

## Client HTTP:

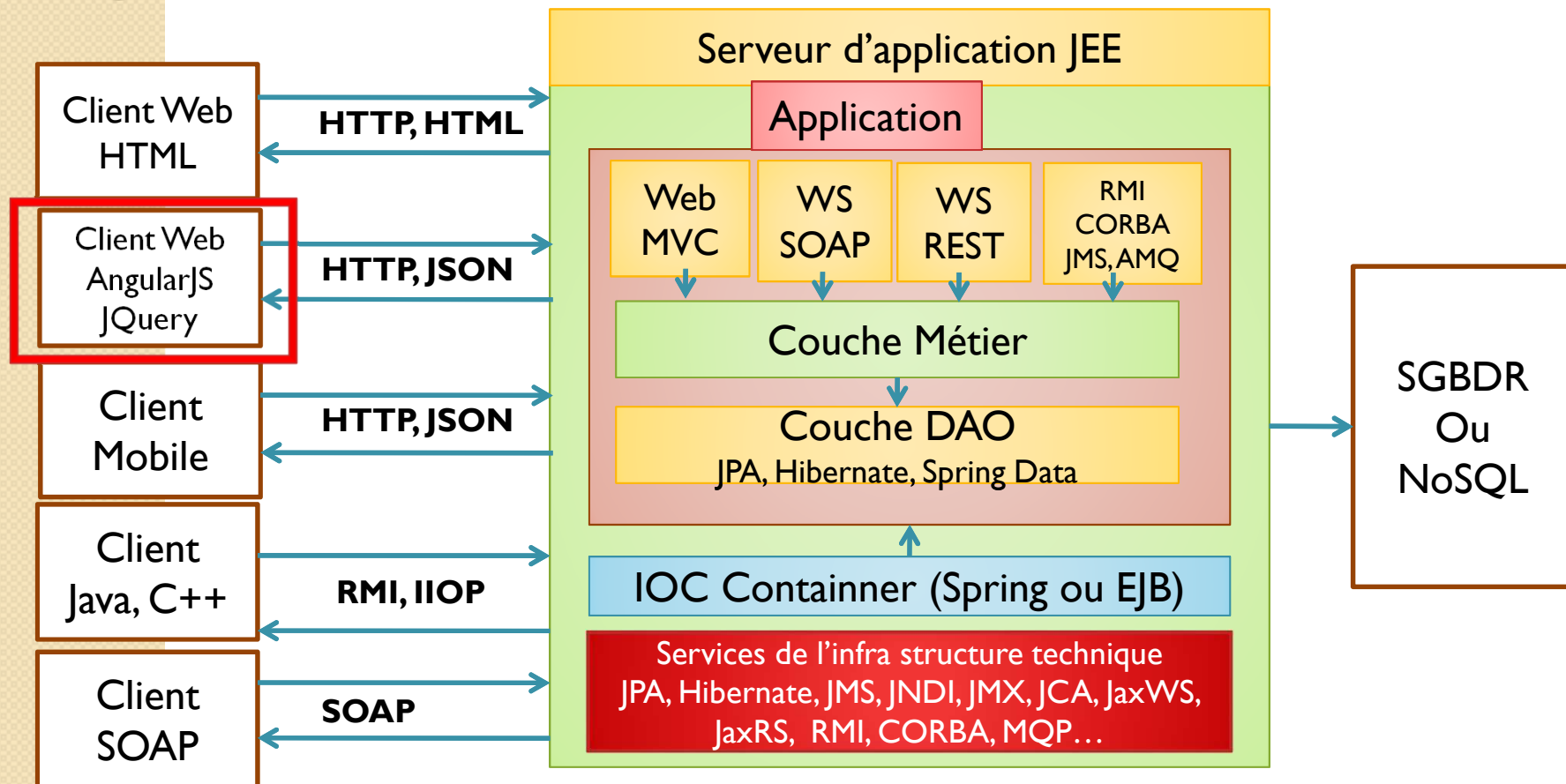
- Envoie les requête HTTP à la couche Web de l'application
- Reçois les réponse HTTP contenant du HTML, CSS et Java Script
- Affiche la Page Web (HTML, CSS, Java Script)
- Dans ce scénario, tout est généré coté serveur



# Architecture d'une application

## Client Web (Riche):

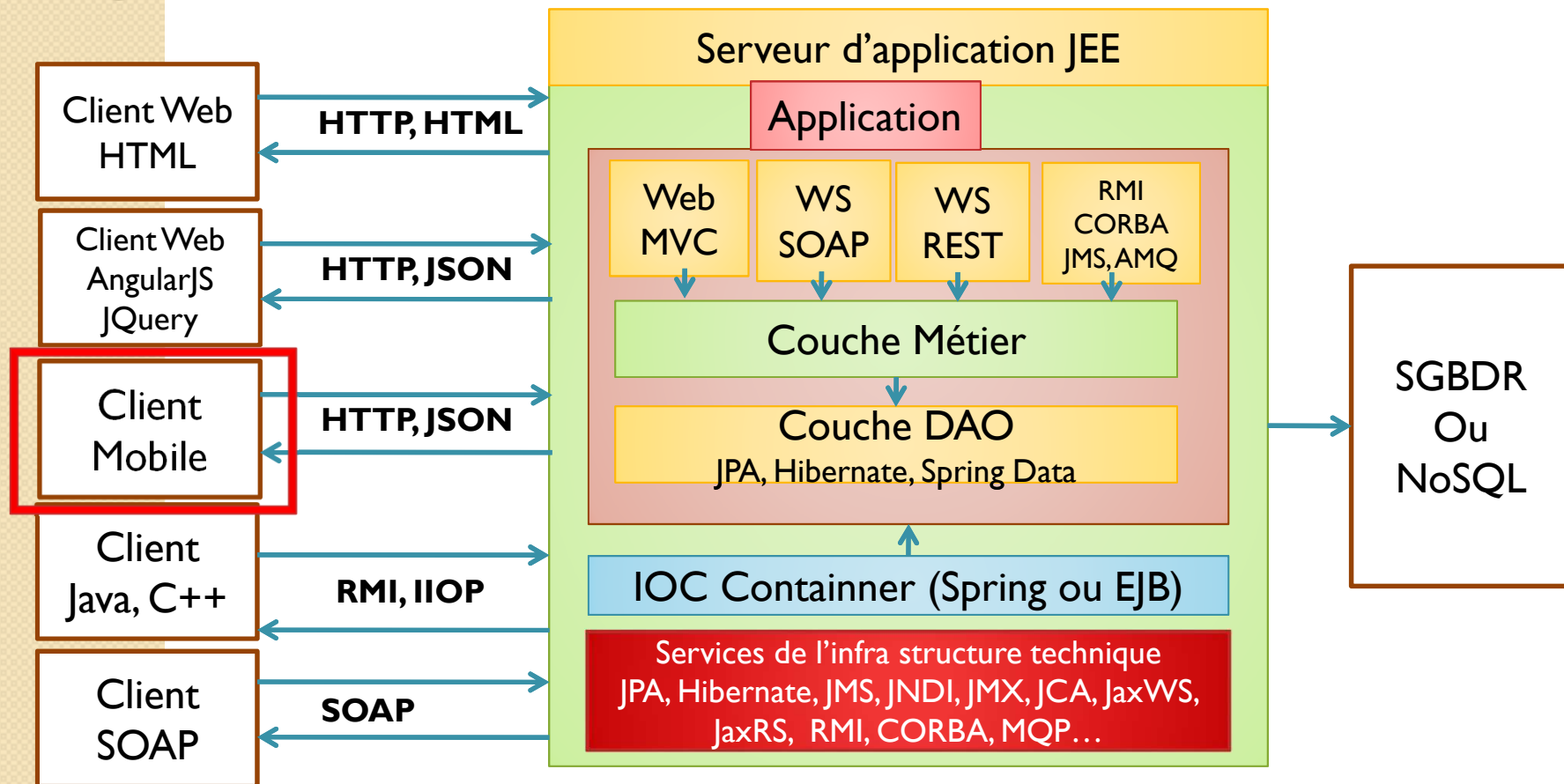
- Envoie les requête HTTP Ajax au service REST en utilisant Java Script
- Ce dernier retourne une réponse HTTP qui contient des données au format JSON
- Affiche ces données JSON dans une page HTML.
- Dans ce cas c'est le client HTTP qui s'occupe de la présentation
- Souvent on utilise un Framework Java Script comme AngularJS ou JQuery



# Architecture d'une application

## Client Mobile:

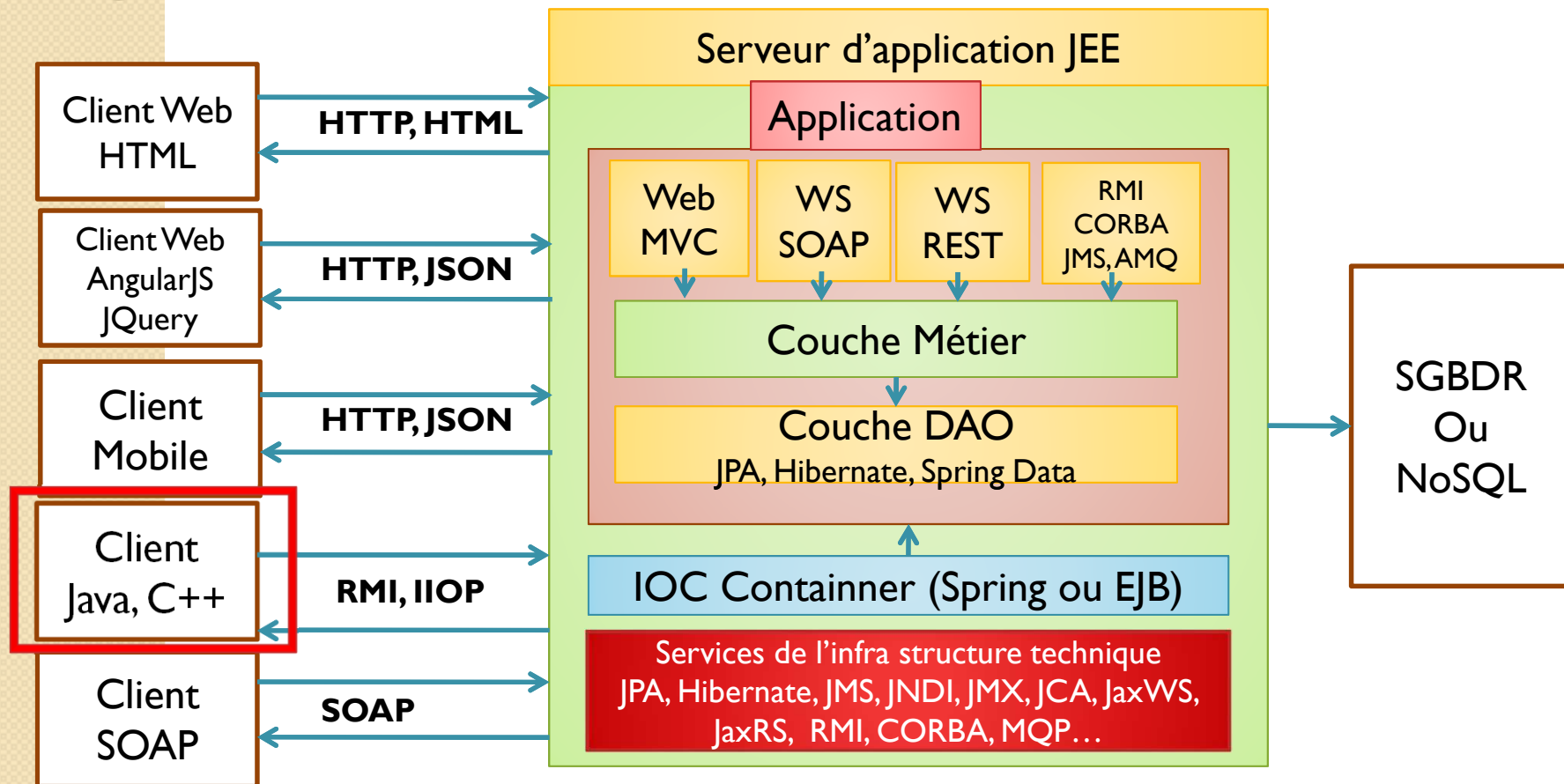
- Envoie les requête HTTP au service REST
- Ce dernier retourne une réponse HTTP qui contient des données au format JSON
- Affiche ces données JSON dans l'inteface de l'application mobile



# Architecture d'une application

## Client Lourd Java, C++, ....:

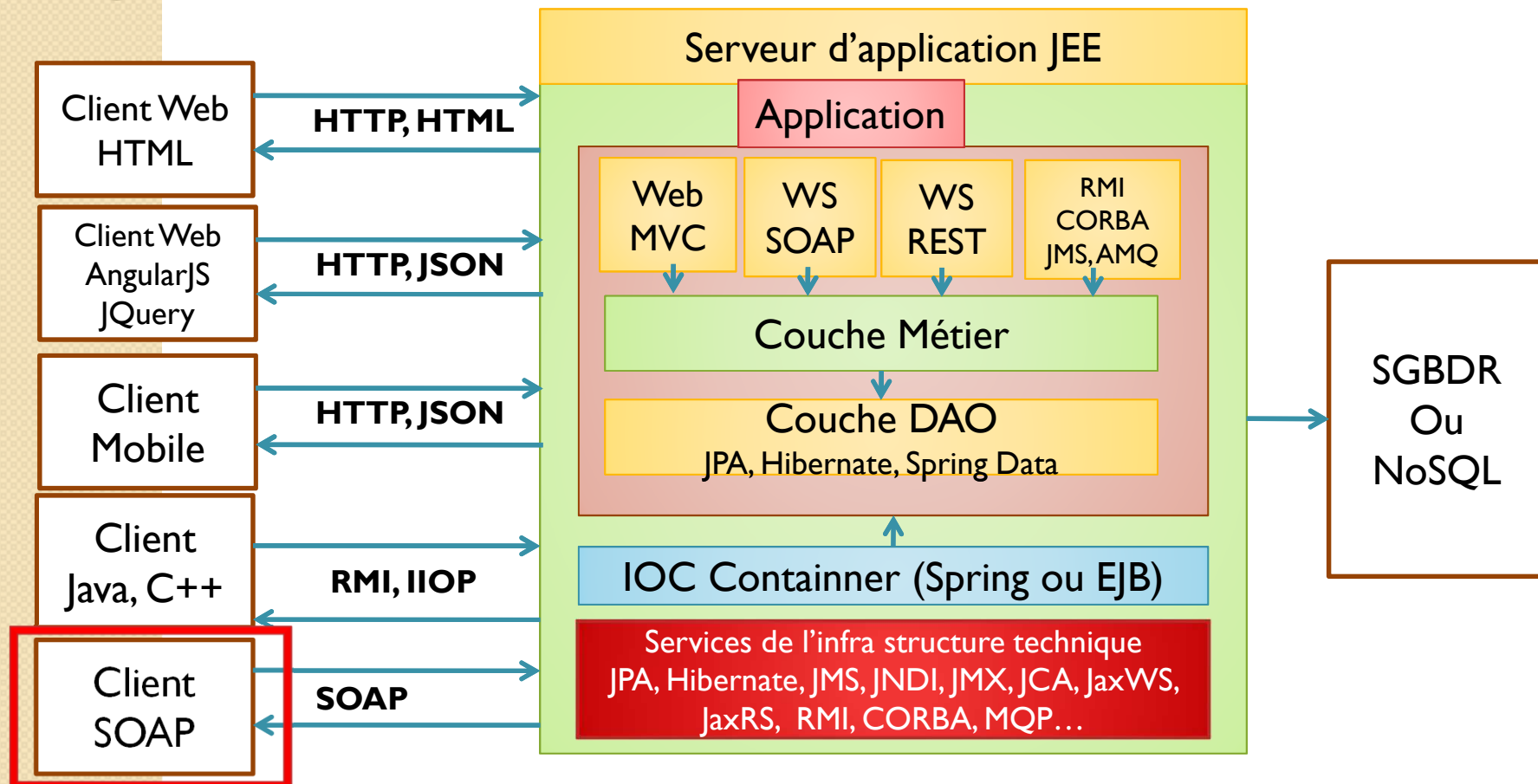
- Un client Java peut communiquer avec un autre objet Java du en utilisant le middleware RMI
- Un Client écrit avec un autre langage de programmation peut communiquer avec l'application via le middleware CORBA




# Architecture d'une application

## Client SOAP:

- Au lieu de CORBA, une application quelconque peut également communiquer à distance avec le web service en utilisant le protocole SOAP (HTTP+XML)





# Inversion de contrôle ou Injection de dépendances

# Rappels de quelques principes de conception

- Une application qui n'évolue pas meurt.
- Une application doit être fermée à la modification et ouverte à l'extension.
- Une application doit s'adapter aux changements
- ...
- Comment Créer une application fermée à la modification et ouverte à l'extension ?
- Comment faire l'injection des dépendances ?
- C'est quoi le principe de l'inversion de contrôle ?

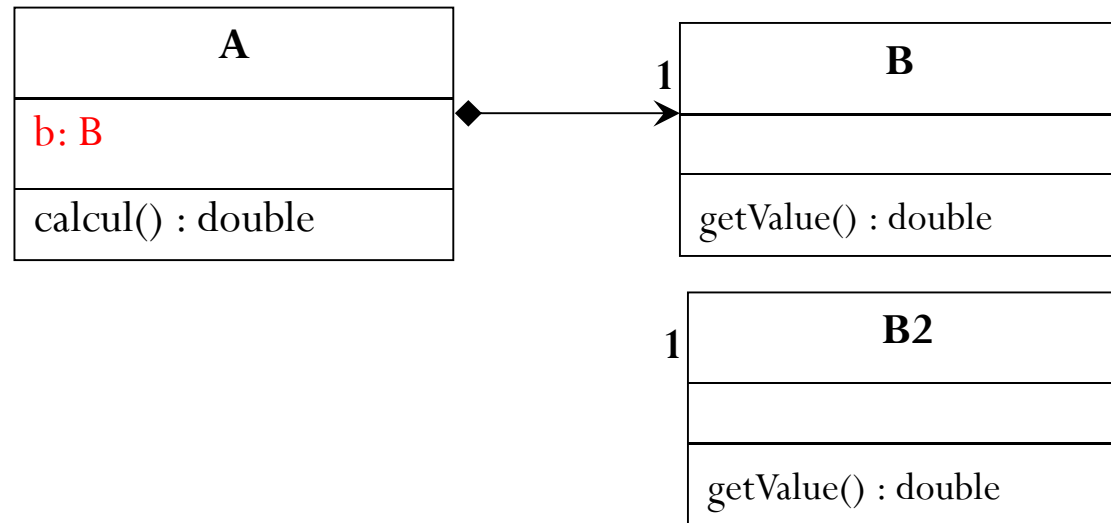


# Couplage Fort et Couplage faible



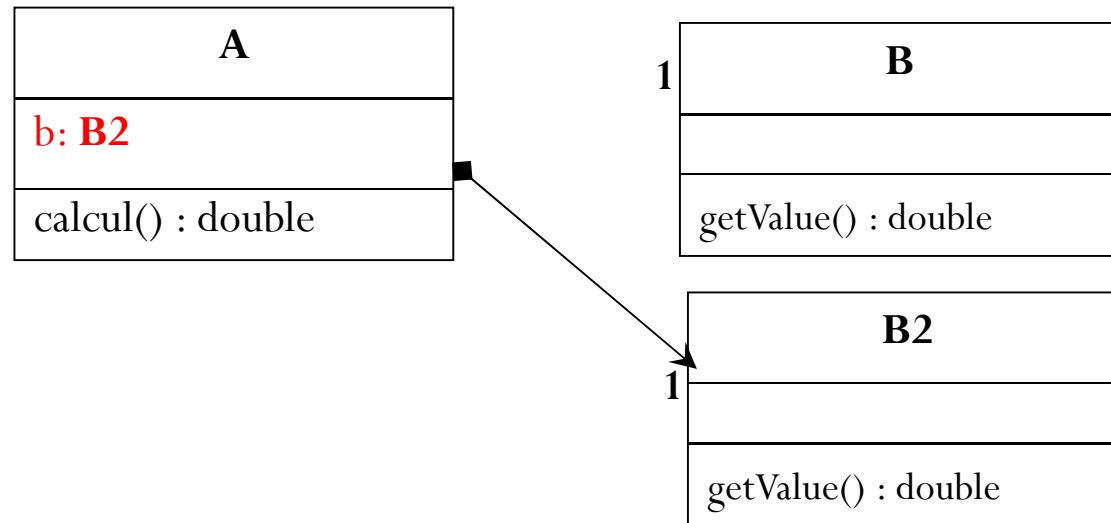
# Couplage fort

- Quand une classe A est liée à une classe B, on dit que la classe A est fortement couplée à la classe B.
- La classe A ne peut fonctionner qu'en présence de la classe B.
- Si une nouvelle version de la classe B (soit B2), est créée, on est obligé de modifier dans la classe A.
- Modifier une classe implique:
  - Il faut disposer du code source.
  - Il faut recompiler, déployer et distribuer la nouvelle application aux clients.
  - Ce qui engendre un cauchemar au niveau de la maintenance de l'application

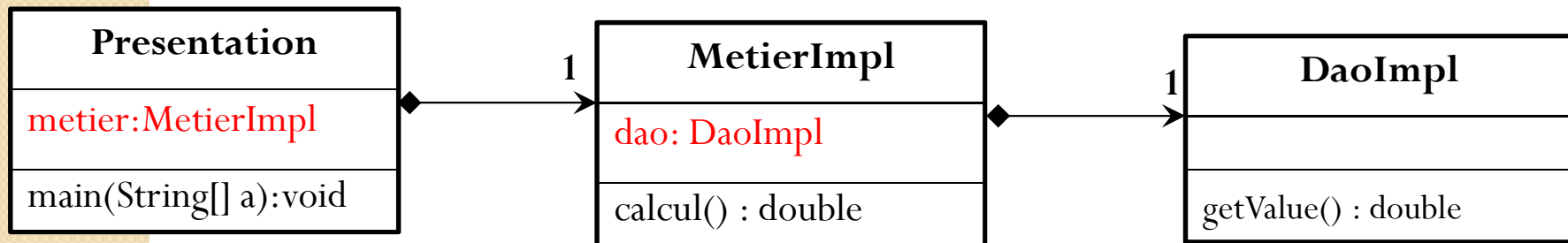


# Couplage fort

- Quand une classe A est liée à une classe B, on dit que la classe A est fortement couplée à la classe B.
- La classe A ne peut fonctionner qu'en présence de la classe B.
- Si une nouvelle version de la classe B (soit B2), est créée, on est obligé de modifier dans la classe A.
- Modifier une classe implique:
  - Il faut disposer du code source.
  - Il faut recompiler, déployer et distribuer la nouvelle application aux clients.
  - Ce qui engendre un cauchemar au niveau de la maintenance de l'application



# Exemple de couplage fort



```
package metier;
import dao.DaoImpl;
public class MetierImpl {
    private DaoImpl dao;
    public MetierImpl() {
        dao=new DaoImpl();
    }
    public double calcul() {
        double nb=dao.getValue();
        return 2*nb;
    }
}
```

```
package dao;
public class DaoImpl {
    public double getValue() {
        return (5);
    }
}
```

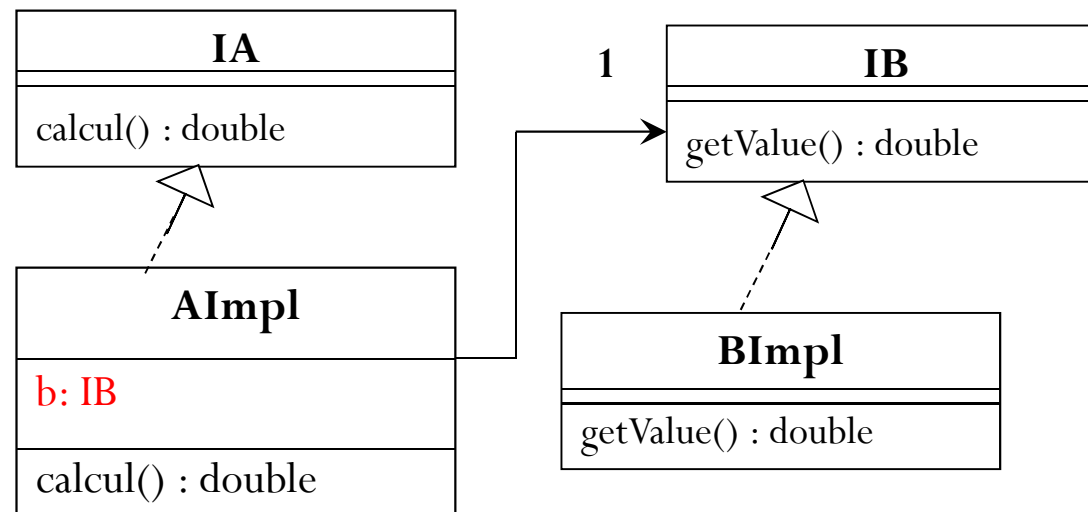
```
package pres;
import metier.MetierImpl;
public class Presentation {
    private static MetierImpl metier;
    public static void main(String[]
        args) {
        metier=new MetierImpl();
        System.out.println(metier.calcul());
    }
}
```

# Problèmes du couplage fort

- Dans l'exemple précédent, les classes MetierImpl et DaoImpl sont liées par un couplage fort. De même pour les classe Presentation et MetierImpl
- Ce couplage fort n'a pas empêché de résoudre le problème au niveau fonctionnel.
- Mais cette conception nous ne a pas permis de créer une application fermée à la modification et ouverte à l'extension.
- En effet, la création d'une nouvelle version de la méthode getValue() de la classe DaoImpl, va nous obliger d'éditer le code source de l'application aussi bien au niveau de DaoImpl et aussi MetierImpl.
- De ce fait nous avons violé le principe « une application doit être fermée à la modification et ouverte à l'exetension »
- Nous allons voir que nous pourrons faire mieux en utilisant le couplage faible.

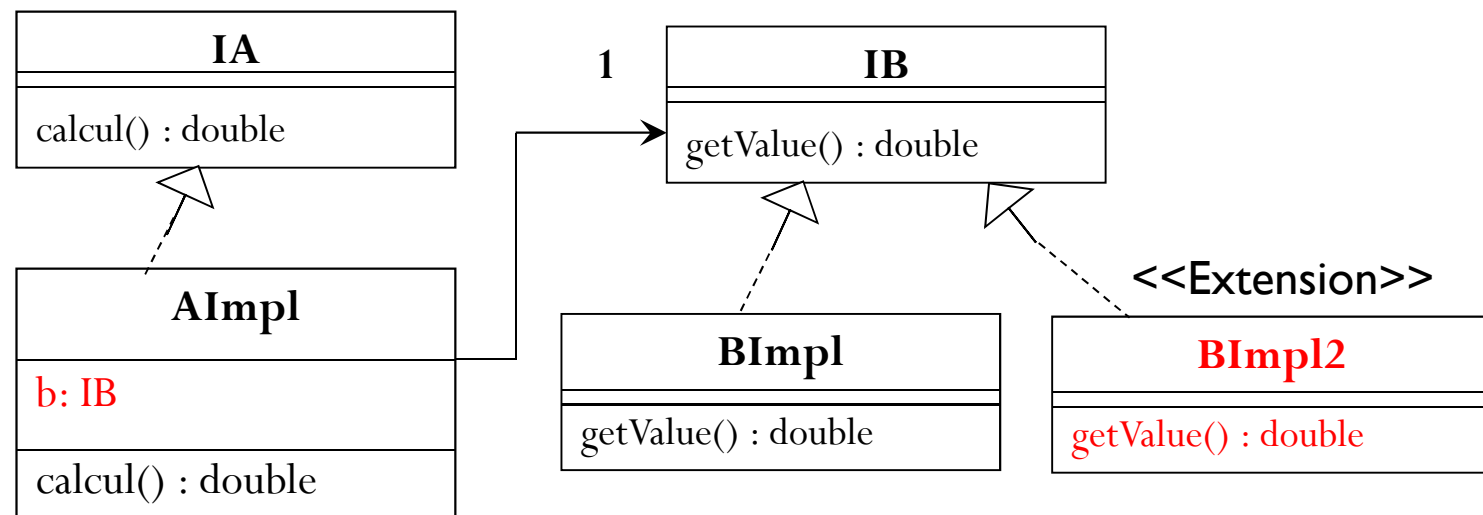
# Couplage Faible.

- Pour utiliser le couplage faible, nous devons utiliser les interfaces.
- Considérons une classe A qui implémente une interface IA, et une classe B qui implémente une interface IB.
- Si la classe A est liée à l'interface IB par une association, on dit que la classe A et la classe B sont liées par un couplage faible.
- Cela signifie que la classe B peut fonctionner avec n'importe quelle classe qui implémente l'interface IA.
- En effet la classe B ne connaît que l'interface IA. De ce fait n'importe quelle classe implémentant cette interface peut être associée à la classe B, sans qu'il soit nécessaire de modifier quoi que se soit dans la classe B.
- Avec le couplage faible, nous pourrions créer des application fermée à la modification et ouvertes à l'extension.

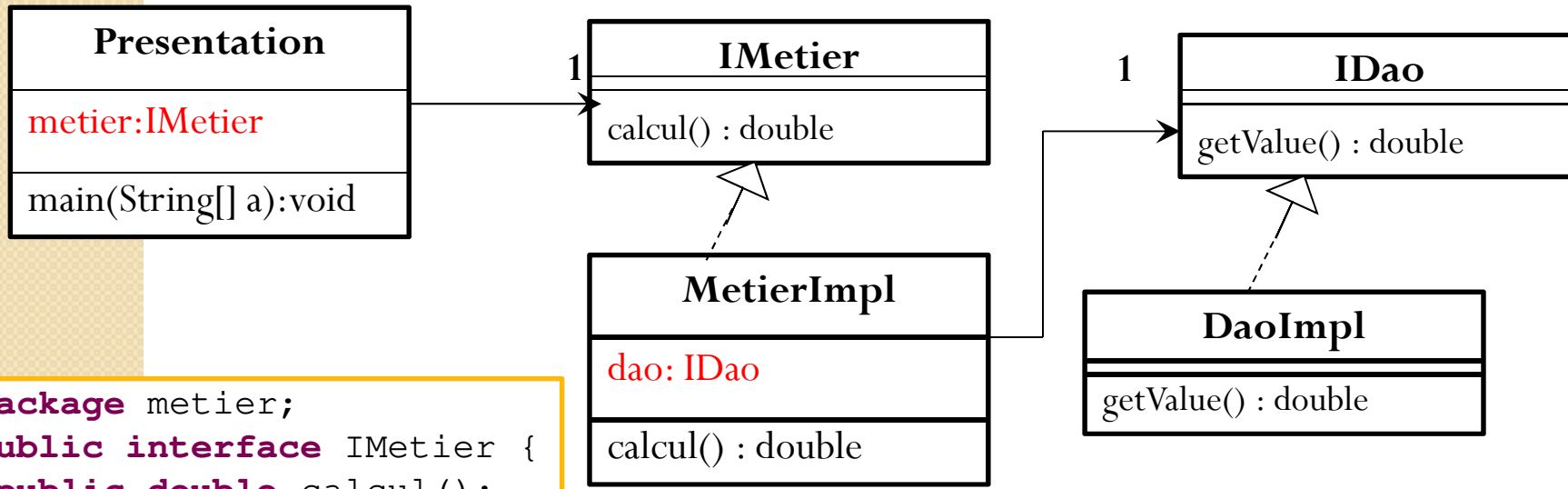


# Couplage Faible.

- Pour utiliser le couplage faible, nous devons utiliser les interfaces.
- Considérons une classe A qui implémente une interface IA, et une classe B qui implémente une interface IB.
- Si la classe A est liée à l'interface IB par une association, on dit que la classe A et la classe B sont liées par un couplage faible.
- Cela signifie que la classe B peut fonctionner avec n'importe quelle classe qui implémente l'interface IA.
- En effet la classe B ne connaît que l'interface IA. De ce fait n'importe quelle classe implémentant cette interface peut être associée à la classe B, sans qu'il soit nécessaire de modifier quoi que se soit dans la classe B.
- Avec le couplage faible, nous pourrions créer des application fermée à la modification et ouvertes à l'extension.



# Exemple de coupage faible



```
package metier;
public interface IMetier {
    public double calcul();
}
```

```
package metier;
import dao.IDao;
public class MetierImpl
    implements IMetier {
    private IDao dao;
    public double calcul() {
        double nb=dao.getValue();
        return 2*nb;
    }
    // Getters et Setters
}
```

```
package dao;
public interface IDao {
    public double getValue();
}
```

```
package dao;
public class DaoImpl implements IDao {
    public double getValue() {
        return 5;
    }
}
```

# Injection des dépendances

- Injection par instantiation statique :

```
import metier.MetierImpl;
import dao.DaoImpl;
public class Presentation {
public static void main(String[] args) {
    DaoImpl dao=new DaoImpl();
    MetierImpl metier=new MetierImpl();
    metier.setDao(dao);
    System.out.println(metier.calcul());
}
}
```



# Injection des dépendances

- Injection par instanciation dynamique par réflexion :
  - Fichier texte de configuration : config.txt

```
ext.DaoImp  
metier.MetierImpl
```

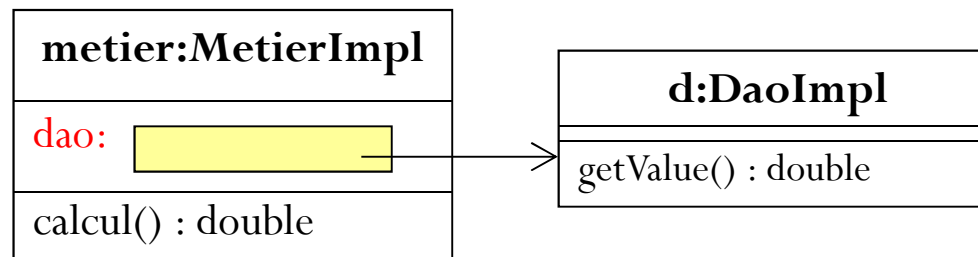
```
import java.io.*;import java.lang.reflect.*;  
import java.util.Scanner; import metier.IMetier;  
import dao.IDao;  
public class Presentation {  
    public static void main(String[] args) {  
        try {  
            Scanner scanner=new Scanner(new File("config.txt"));  
            String daoClassname=scanner.next();  
            String metierClassName=scanner.next();  
            Class cdao=Class.forName(daoClassname);  
            IDao dao= (IDao) cdao.newInstance();  
            Class cmetier=Class.forName(metierClassName);  
            IMetier metier=(IMetier) cmetier.newInstance();  
            Method meth=cmetier.getMethod("setDao",new Class[]{IDao.class});  
            meth.invoke(metier, new Object[]{dao});  
            System.out.println(metier.calcul());  
        } catch (Exception e) { e.printStackTrace(); }  
    }  
}
```

# Injection des dépendances avec Spring.

- L'injection de dépendances est souvent la base de tout programme moderne.
- L'idée en résumé est de déporter la responsabilité de la liaison des composants du programme dans un framework afin de pouvoir facilement changer ces composants ou leur comportement.
- Parmi les leaders du marché Java, il y a Spring IoC, Guice, Dagger ou encore le standard « Java EE » CDI qui existe depuis Java EE 6.
- Spring IOC commence par lire un fichier XML qui déclare quelles sont différentes classes à instancier et d'assurer les dépendances entre les différentes instances.
- Quand on a besoin d'intégrer une nouvelle implémentation à une application, il suffirait de la déclarer dans le fichier xml de beans spring.

# Injection des dépendances dans une application java standard

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-
    2.0.dtd" >
<beans>
  <bean id="d" class="dao.DaoImpl2"></bean>
  <bean id="metier" class="metier.MetierImpl">
    <property name="dao" ref="d"></property>
  </bean>
</beans>
```



# Injection des dépendances dans une application java standard

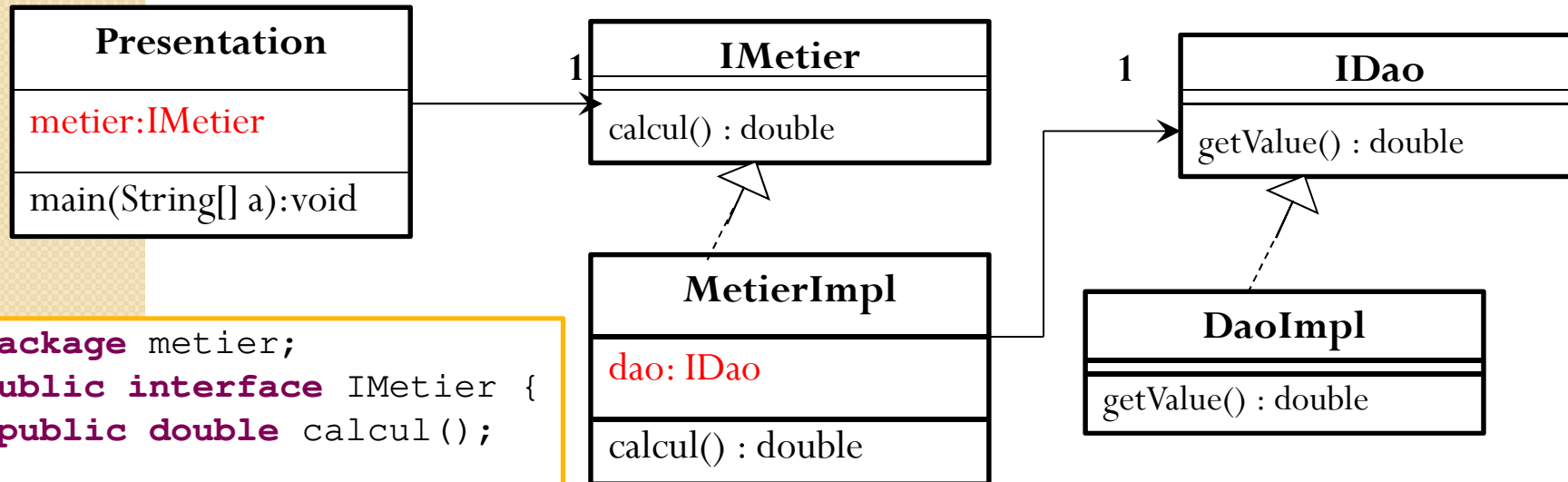
```
package pres;

import metier.IMetier;

import org.springframework.context.support.*;

public class Presentation {
    public static void main(String[] args) {
        ApplicationContext context=new
        ClassPathXmlApplicationContext("spring-ioc.xml");
        IMetier metier=context.getBean(IMetier.class);
        System.out.println(metier.calcul());
    }
}
```

# Injection par annotations Spring au lieu de XML



```
package metier;
public interface IMetier {
    public double calcul();
}
```

```
package metier;
import dao.IDao;
@Component("metier")
public class MetierImpl
    implements IMetier {
    @Autowired
    @Resource("dao")
    private IDao dao;
    public double calcul() {
        double nb=dao.getValue();
        return 2*nb;
    }
}
```

```
package dao;
public interface IDao {
    public double getValue();
}
```

```
package dao;
@Component("dao")
public class DaoImpl implements IDao {
    public double getValue() {
        return 5;
    }
}
```

# Injection des dépendances dans une application java standard

```
package pres;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import metier.IMetier;

public class Presentation {
    public static void main(String[] args) throws Exception {
        ApplicationContext ctx=new AnnotationConfigApplicationContext("dao","metier");
        //ApplicationContext ctx=new ClassPathXmlApplicationContext("config.xml");
        IMetier metier=ctx.getBean(IMetier.class);
        System.out.println(metier.calcul());
    }
}
```

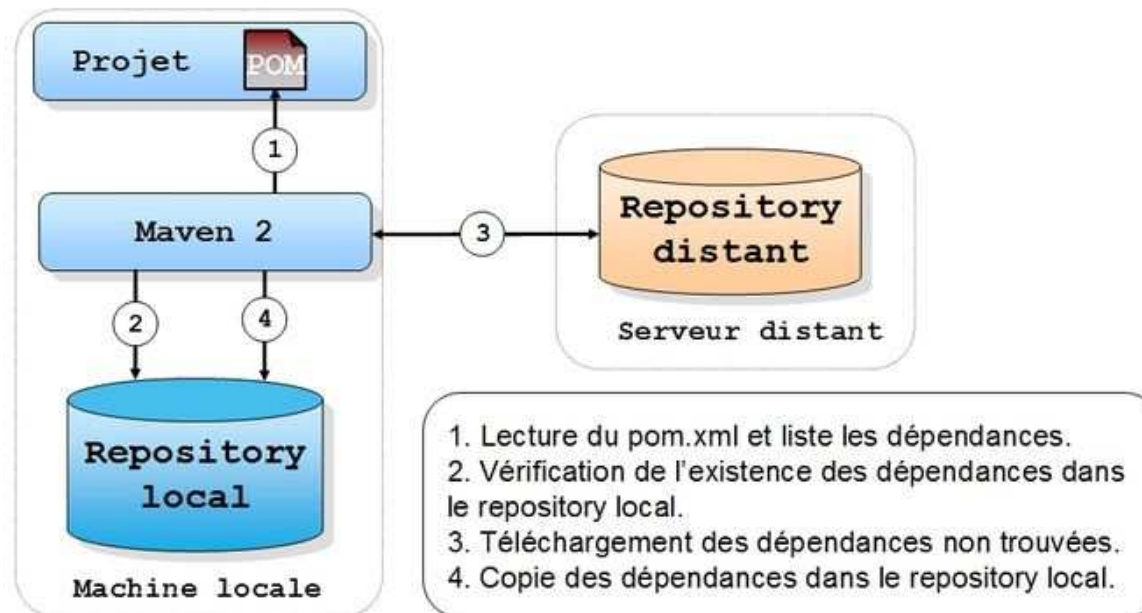
# Maven

- **Maven**, géré par l'organisation *Apache Software Foundation*. (*Jakarta Project*), est un **outil pour la gestion et l'automatisation de production des projets logiciels Java en général et Java EE en particulier**.
- L'objectif recherché est de
  - produire un logiciel à partir de ses sources,
  - en optimisant les tâches réalisées à cette fin
  - et en garantissant le bon ordre de fabrication.
    - **Compiler, Tester, Contrôler, produire les packages livrables**
    - **Publier la documentation et les rapports sur la qualité**
- **Apports :**
  - Simplification du processus de construction d'une application
  - Fournit les bonnes pratiques de développement
  - Tend à uniformiser le processus de construction logiciel
  - Vérifier la qualité du code
  - Faciliter la maintenance d'un projet

# Maven : POM

- Maven utilise un paradigme connu sous le nom de **Project Object Model (POM)** afin de :
  - Décrire un projet logiciel,
  - Ses dépendances avec des modules externes
  - et l'ordre à suivre pour sa production.
- Il est livré avec un grand nombre de tâches (**GOLS**) prédéfinies, comme la compilation du code Java ou encore sa modularisation.

## Gestion des dépendances par Maven 2

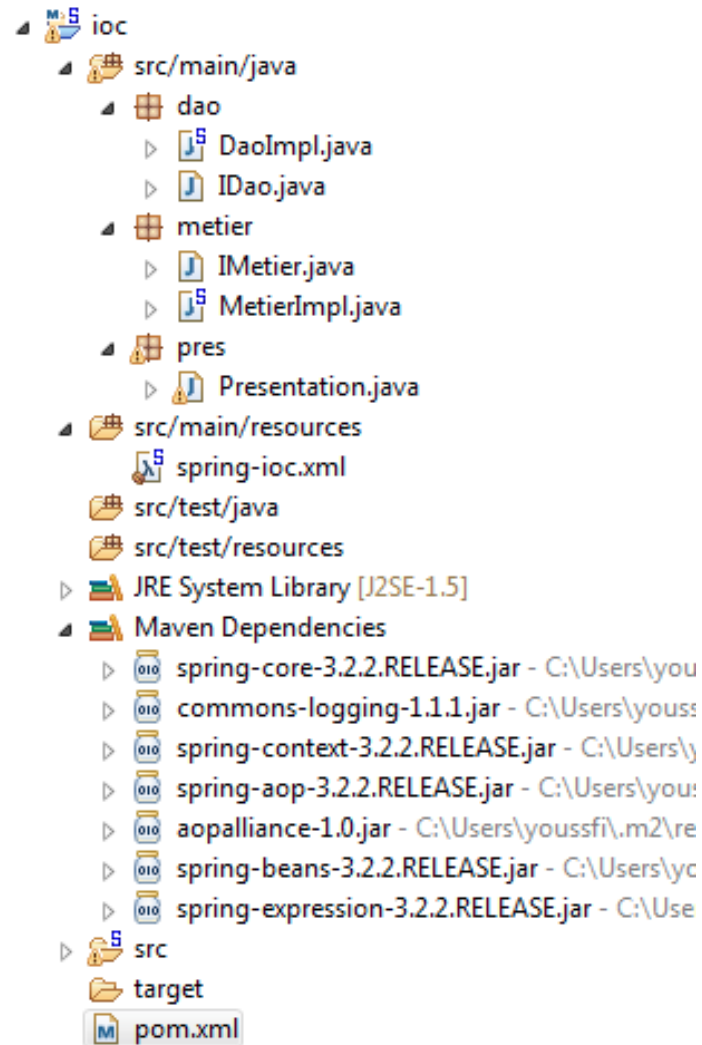




# Maven dependencies

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>3.2.2.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>3.2.2.RELEASE</version>
  </dependency>
</dependencies>
```

# Structure du projet



# Injection des dépendances dans une application web

- Dans une application web, SpringIOC est appelé au démarrage du serveur en déclarant le listener ContextLoaderListener dans le fichier **web.xml**

```
<context-param>
```

```
    <param-name>contextConfigLocation</param-name>
```

```
    <param-value>/WEB-INF/spring-beans.xml</param-value>
```

```
</context-param>
```

```
<listener>
```

```
    <listener-class>
```

```
        org.springframework.web.context.ContextLoaderListener
```

```
    </listener-class>
```

```
</listener>
```

- Dans cette déclaration, ContextLoaderListener est appelé par Tomcat au moment du démarrage de l'application. Ce listener cherchera le fichier de beans spring « spring-beans.xml » stocké dans le dossier WEB-INF. ce qui permet de faire l'injection des dépendances entre MetierImpl et DaoImpl



# CDI : Context Dependency Injection

- CDI (Context and Dependency Injection) est une spécification destinée à standardiser l'injection de dépendances et de contextes, au sein de la plateforme Java et plus particulièrement Java EE.
- Intégrée à la spécification Java EE 6, sa version 1.0 est sortie en décembre 2009 et a été suivie des versions 1.1 (mai 2013) et 1.2 (avril 2014). Son élaboration a été le fruit des JSR 299 et 346.
- Le but de cette spécification est de définir les API et les comportements associés en ce qui concerne ce qu'on appelle communément l'injection de dépendances (inversion de contrôle).
- Très rapidement, le but de cette dernière est d'avoir un couplage lâche entre nos classes. En d'autres termes, définir un contrat entre les beans, mais faciliter le remplacement d'un maillon de la chaîne très facilement sans avoir à faire les liens et l'initialisation soi-même.



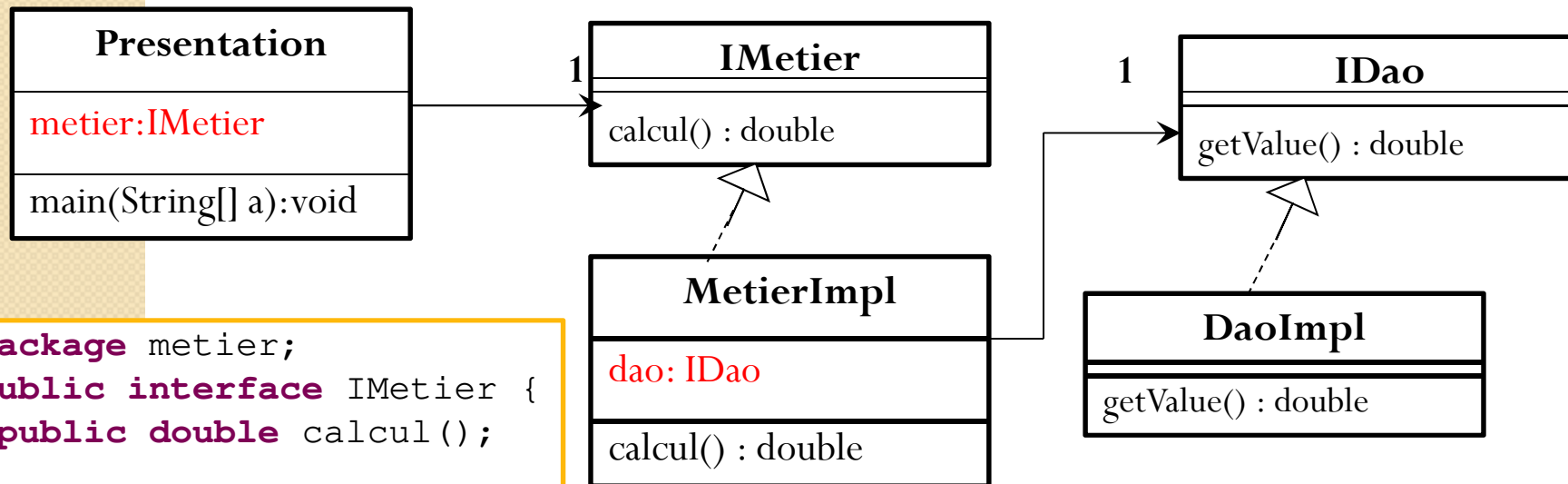
# Principales implémentations de CDI

- CDI a trois implémentations principales :
  - **Weld** : implémentation de référence de CDI. Utilisé dans GlassFish, JBoss (AS et WildFly) ;
  - **OpenWebBeans** : implémentation Apache. Utilisé dans TomEE, OpenEJB, Geronimo, WebSphere ;
  - **CanDi** : implémentation Caucho. Utilisé dans Resin.

# CDI Maven dependencies

```
<dependencies>  
  <dependency>  
    <groupId>org.jboss.weld.se</groupId>  
    <artifactId>weld-se-core</artifactId>  
    <version>2.4.1.Final</version>  
  </dependency>  
</dependencies>
```

# Injection par annotations Spring au lieu de XML



```
package metier;
public interface IMetier {
    public double calcul();
}
```

```
package metier;
import dao.IDao;
@Singleton
@Named
public class MetierImpl
    implements IMetier {
    @Inject
    private IDao dao;
    public double calcul() {
        double nb=dao.getValue();
        return 2*nb;
    }
}
```

```
package dao;
public interface IDao {
    public double getValue();
}
```

```
package dao;
@Named
public class DaoImpl implements IDao {
    public double getValue() {
        return 5;
    }
}
```

# Injection des dépendances dans une application java standard

```
package cdi;
import org.jboss.weld.environment.se.Weld;
import org.jboss.weld.environment.se.WeldContainer;
import cdi.metier.IMetier;
public class App {

    public static void main(String[] args) {
        Weld weld = new Weld();
        WeldContainer container = weld.initialize();
        IMetier metier=container.select(IMetier.class).get();
        System.out.println(metier.calcul());
    }
}
```

## META-INF/beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">

</beans>
```