

L'objectif de ce tutoriel pas à pas est la réalisation d'une architecture distribuée basée sur Java EE 7 et le serveur d'application Payara Server.
Ce n'est en aucun cas un cours.

Partie 1 : Création de services Web SOAP et RESTful et d'un client de services web

Création des domaines store et middleware

Vous allez d'abord créer les 2 domaines utilisés pour cette première partie de ce tutoriel.

store : l'application web cliente sera déployée dans ce domaine.

middleware : domaine dans lequel seront déployés les services web.

Il est fortement conseillé de définir un mot de passe différent pour chaque domaine.

Pour créer ces 2 domaines vous allez utiliser la commande *asadmin*.

Vous allez utiliser la commande *asadmin* avec l'argument *portbase* qui permet de déterminer une base à partir de laquelle les ports du domaine seront attribués.

1. Création du domaine **store** [avec portbase 10 000] :

```
asadmin --user admin create-domain --portbase 10000 --savemasterpassword=true store
```

Lors de la création du domaine *store* les ports seront assignés en se basant sur le port 10 000.

Saisissez un mot de passe.

Choisissez une politique de mot de passe simple et intuitive pour l'administration de vos domaines : par exemple pour le domaine *store*, utilisez le mot de passe **astore** (le **a** c'est pour admin) pour le compte *admin*.

Pour le *mot de passe maître* (*master password*), appuyez juste sur entrée : le mot de passe par défaut **changeit** sera assigné.

La sortie devrait afficher :

```
Using port 10048 for Admin.  
Using port 10080 for HTTP Instance.  
Using port 10076 for JMS.  
Using port 10037 for IIOP.  
Using port 10081 for HTTP_SSL.  
Using port 10038 for IIOP_SSL.  
Using port 10039 for IIOP_MUTUALAUTH.  
Using port 10086 for JMX_ADMIN.
```

Using port 10066 for OSGI_SHELL.

Using port 10009 for JAVA_DEBUGGER.

Distinguished Name of the self-signed X.509 Server Certificate is:

[CN=bank,OU=GlassFish,O=Oracle Corporation,L=Santa Clara,ST=California,C=US]

Distinguished Name of the self-signed X.509 Server Certificate is:

[CN=bank-instance,OU=GlassFish,O=Oracle Corporation,L=Santa Clara,ST=California,C=US]

Domain store created.

Domain store admin port is 10048.

Domain store admin user is "admin".

Command create-domain executed successfully.

2. Répétez l'opération pour créer le domaine *middleware*, en attribuant un port de base différent, autrement dit une plage de ports d'écoute différente. Pour le mot de passe maître, laissez le mot de passe par défaut changeit.

Création du domaine **middleware** [avec portbase 11 000]:

```
asadmin --user admin create-domain --portbase 11000 --savemasterpassword=true middleware
```

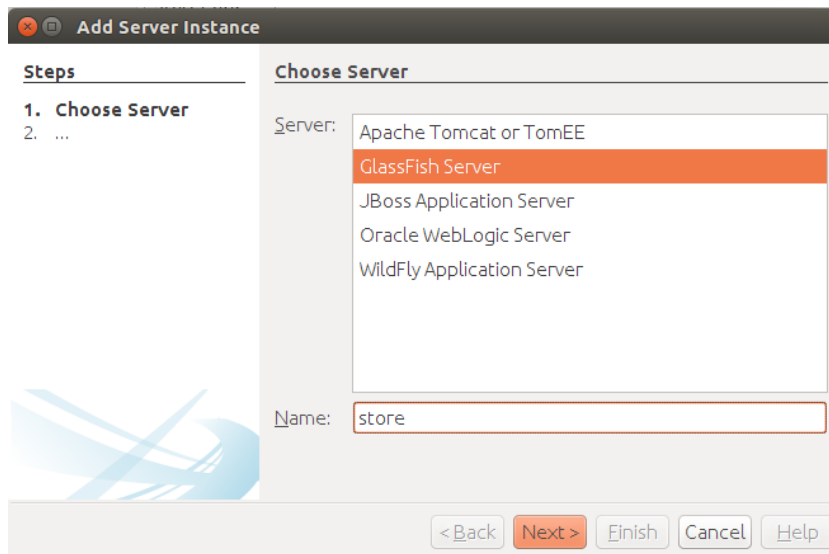
Utilisez par exemple le mot de passe **amiddle**.

Vérifiez que les dossiers correspondant à chaque domaine ont bien été créés dans <dossier_installation_payara>/glassfish/domains/

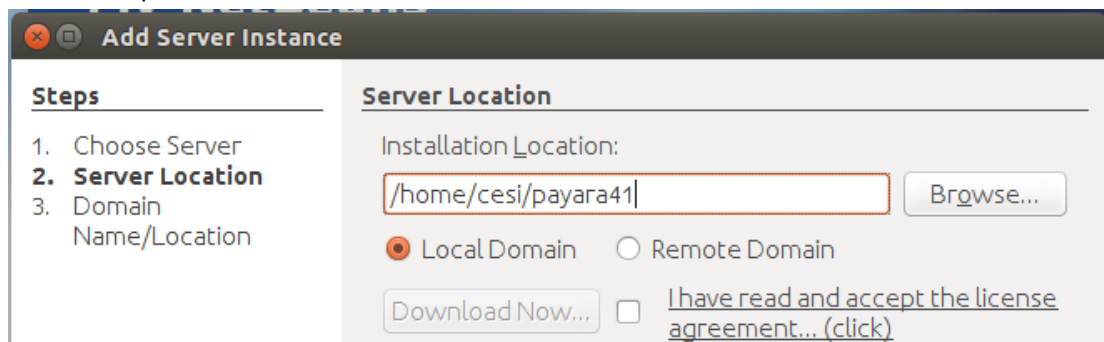
Vous devriez trouver un dossier store et middleware.

Maintenant il s'agit de référencer dans NetBeans les domaines créés. Il faut ouvrir l'onglet Services (Rappel : l'onglet Services est accessibles depuis le menu Windows de NetBeans).

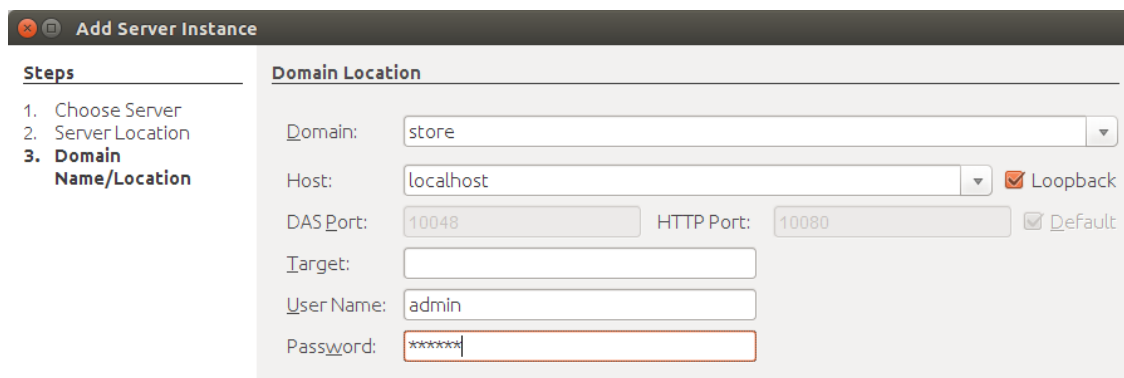
3. Ajouter la référence au domaine store :
 - a. Clic <droite> sur le nœud Servers puis Add Server...
 - b. Sélectionnez GlassFish Server, nommez la référence store puis cliquez Next>



- c. Cliquez sur Browse... pour sélectionner le dossier de votre installation Payara et cliquez sur Next>



- d. Dans le menu déroulant, sélectionnez le domaine **store**, saisissez le nom utilisateur admin et le mot de passe pour le compte admin (**astore** dans notre cas), enfin cliquez sur Finish.



Recommencez l'étape 3 pour le domaine middleware.

Domaine middleware :

Nom d'instance : middleware

Sélection du dossier d'installation de Payara comme précédemment

Domaine sélectionné : middleware

Nom utilisateur : admin / mot de passe amiddle dans notre cas.

Au final vous avez deux domaines qui peuvent s'exécuter en même temps.



4. Démarrez les domaines depuis NetBeans pour vérifier que tout fonctionne correctement.

Si les domaines ont correctement été lancés, vous pouvez les stopper.

Rappel : clic <droite> sur le nœud du domaine et *start*.

Conseil : créez tous vos projets à la racine d'un même dossier dont le chemin ne comporte pas d'espace ou de caractères spéciaux.

A partir de maintenant, soyez très attentifs aux domaines dans lesquels seront déployés les différents projets.

Créer un web service SOAP avec le standard JAX-WS

Domaine middleware

Objectif : exposer un Stateless session bean en tant que web service

5. Dans NetBeans, Créer un nouveau projet Java Maven de type **EJB Module**.
6. Nommez-le bankFacade-ejb.
7. Localisez-le dans le dossier racine de vos projets.
8. Spécifiez le Group Id : com.bank
9. Donnez un nom de paquetage par défaut : com.bank.paymentmgmt.facade

Steps

1. Choose Project
2. **Name and Location**
3. Settings

Name and Location

Project Name:	<input type="text" value="bankFacade-ejb"/>	
Project Location:	<input type="text" value="/home/cesi/javaProjects/v2017/v1"/>	<input data-bbox="1369 450 1493 488" type="button" value="Browse..."/>
Project Folder:	<input type="text" value="/home/cesi/javaProjects/v2017/v1/bankFacade-ejb"/>	
Artifact Id:	<input type="text" value="bankFacade-ejb"/>	
Group Id:	<input type="text" value="com.bank"/>	
Version:	<input type="text" value="1.0-SNAPSHOT"/>	
Package:	<input type="text" value="com.bank.paymentmgmt.facade"/>	(Optional)

10. Associez-le à **middleware**.

11. Validez que la version Java EE est bien Java EE 7 et cliquez sur Finish.

Steps

1. Choose Project
2. Name and Location
3. **Settings**

Settings

Server:	<input type="text" value="middleware"/>	<input data-bbox="1417 949 1493 987" type="button" value="Add..."/>
Java EE Version:	<input type="text" value="Java EE 7"/>	

La création du module, entraîne le téléchargement des dépendances Maven.

Modifiez le fichier pom.xml situé sous Project Files :

- Donnez un nom final à l'archive : bankFacade-ejb
- Changez la version Java utilisée par Maven. Java 8 est utilisé.
- Précisez au niveau du plugin EJB que c'est la version 3.2 (Java EE 7) qui est utilisée.

Voici un extrait du pom.xml avec les modifications en gras :

```
...
<dependencies>
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-api</artifactId>
    <version>7.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>

<build>
  <finalName>bankFacade-ejb</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
```

```
<configuration>
  <source>1.8</source>
  <target>1.8</target>
  <compilerArguments>
    <endorseddirs>${endorsed.dir}</endorseddirs>
  </compilerArguments>
</configuration>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-ejb-plugin</artifactId>
  <version>2.3</version>
  <configuration>
    <ejbVersion>3.2</ejbVersion>
  </configuration>
</plugin> ...
```

Vous allez créer un Session Bean Stateless exposant une vue de type web service SOAP. Vous allez créer une interface Java afin de décrire la vue du web service et ainsi la séparer de l'implémentation fournie par la classe du Session Bean. Cette interface définit donc de manière explicite l'interface du point de terminaison du service (Service Endpoint Interface). C'est pour cela que vous n'allez pas utiliser les outils de création de service web SOAP fourni par NetBeans, car ce dernier vous assistera dans la création d'un Session Bean sans interface dont la classe sera annotée avec @WebService. Cela est tout à fait légal, mais ça ne correspond pas à notre design.

12. Dans le paquetage com.bank.paymentmgmt.facade créez un Session Bean Stateless sans interface nommé BankingServiceBean. Vous pouvez utiliser les fonctionnalités de NetBeans de création de Session Bean.

Steps

1. Choose File Type
2. Name and Location

Name and Location

EJB Name:

Project:

Location:

Package:

Session Type:

- ☒ Stateless
☐ Stateful
☐ Singleton

Create Interface:

- ☐ Local
☐ Remote

Si vous avez généré le squelette du Session Bean via NetBeans vous devriez avoir le code suivant :

```
package com.bank.paymentmgmt.facade;

import javax.ejb.Stateless;
import javax.ejb.LocalBean;

@Stateless
@LocalBean
public class BankingServiceBean {

    // Add business logic below. (Right-click in editor and choose
    // "Insert Code > Add Business Method")
}
```

Vous pouvez supprimer l'annotation @LocalBean. Supprimez aussi le session bean NewSessionBean créé par Maven lors de la création du projet.

13. Toujours dans ce paquetage, créez l'interface Java *BankingServiceEndpointInterface*. Ajoutez dans cette interface la méthode publique d'entête **Boolean createPayment(String ccNumber, Double amount) ;**
14. Spécifiez que la classe *BankingServiceBean* implémente cette interface *BankingServiceEndpointInterface*.
Voici un extrait du squelette de code que vous devriez avoir :

Interface
public interface BankingServiceEndpointInterface {

```

Boolean createPayment(String ccNumber, Double amount) ;
}

Classe du Session Bean

@Stateless
public class BankingServiceBean implements BankingServiceEndpointInterface{

    @Override
    public Boolean createPayment(String ccNumber, Double amount) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}

```

Note : La levée d'exception a été ajoutée automatiquement par NetBeans. Vous pouvez la supprimer.

15. Ajoutez dans le corps de la méthode `createPayment`, le code de test suivant (validant simplement la présence de 10 caractères dans le numéro de CB) :

```

if(ccNumber.length()== 10 ){
    System.out.println("Montant payé : "+amount +" €");
    return true;
} else {
    return false;
}

```

Vous allez maintenant exposer ce Session Bean en tant que service web SOAP. Notez que nous choisissons la génération automatique par le moteur JAX-WS du WSDL en fonction des annotations configurant le service web qui sont spécifiées sur le Session Bean. C'est une approche « Bottom Up » : on part de l'implémentation Java sous-jacente pour générer le contrat exposé sous forme de WSDL.

Remarquez que toutes les annotations JAX-WS que vous allez spécifier résident dans le paquetage *javax.jws*.

16. Annotez l'interface *BankingServiceEndpointInterface* avec

@WebService(name = "BankingEndpoint")

name : le nom du service web. C'est le nom de l'élément XML <wsdl:porttype>

17. Annotez la méthode `createPayment` définie dans l'interface avec

@WebMethod(operationName = "paymentOperation")

operationName : le nom de l'élément <wsdl:operation>

18. Spécifiez toujours au niveau de la méthode définie dans l'interface le nom des paramètres du message SOAP qui sera envoyé au service web :


```
@WebParam(name = "cardNumber")String ccNumber, @WebParam(name = "amountPaid")
```

```
Double amount
```

Profitez-en pour définir aussi le nom de la valeur dans le message de retour en annotant la méthode de l'interface :

```
@WebResult(name = "acceptedPayment")
```

Note : Tous les attributs spécifiés jusqu'ici dans les annotations sont optionnels, mais ils nous permettent de caractériser le WSDL que le moteur JAX-WS va générer.

L'interface du point de terminaison du web service devrait ressembler à ça :

```
package com.bank.paymentmgmt.facade;

import javax.ws.WebMethod;
import javax.ws.WebParam;
import javax.ws.WebService;

@WebService(name = "BankingEndpoint")
public interface BankingServiceEndpointInterface {

    @WebMethod(operationName = "paymentOperation")
    @WebResult(name = "acceptedPayment")
    Boolean createPayment(@WebParam(name="cardNumber") String ccNumber,
        @WebParam(name="amountPaid") Double amount);
}
```

19. Liez ce point de terminaison à l'implémentation représentée par la classe du Session Bean. Pour cela annotez la classe du Session Bean avec

```
@WebService(
    endpointInterface = "com.bank.paymentmgmt.facade.BankingServiceEndpointInterface",
    portName = "BankingPort",
    serviceName = "BankingService"
)
```

endpointInterface : attribut obligatoire lorsque une interface Java est explicitement utilisée comme Service Endpoint Interface (SEI). Cet attribut référence le nom pleinement qualifié de la SEI.

Petit conseil : Comme vous le constatez la chaîne de caractères est longue et le risque d'erreur est d'autant plus grand. Le mieux est donc de faire du copier/coller. Copier & coller donc le nom du paquetage. Ajoutez un point puis copier/coller le nom de l'interface.

portName (optionnel) : le nom du <wsdl:port>.

serviceName (optionnel) : le nom du <wsdl:service>

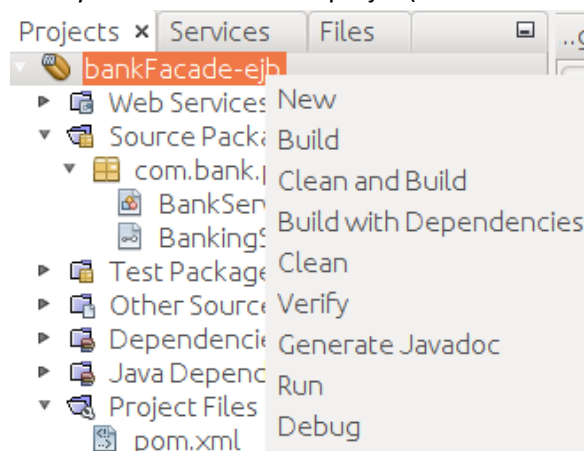
Ces 3 attributs ne peuvent être spécifiés que sur une classe. Ils ne peuvent pas être utilisés en conjonction de `@WebService` annotant une interface Java. L'explication tient au fait que `<wsdl:port>` et `<wsdl:service>` sont des éléments de la partie « implémentation » du WSDL et que `endpointInterface` est un attribut destiné à spécifier le nom de l'interface explicitement définie.

Voici `BankingServiceBean` devrait ressembler à votre classe `BankingServiceBean` :

```
@Stateless
@WebService(
    endpointInterface = "com.bank.paymentmgmt.facade.BankingServiceEndpointInterface",
    portName = "BankingPort",
    serviceName = "BankingService"
)
public class BankingServiceBean implements BankingServiceEndpointInterface{

    @Override
    public Boolean createPayment(String ccNumber, Double amount) {
        if(ccNumber.length()== 10 ){
            System.out.println("Montant payé : "+amount +" €");
            return true;
        } else {
            return false;
        }
    }
}
```

20. nettoyez et construisez le projet (*Clean and Build*) puis exécutez en cliquant sur *Run*.



Si dans la console des logs NetBeans pour le domaine middleware vous avez ce type de message :

Severe: Class com.MAUVAIS_NOM.paymentmgmt.facade.BankingServiceEndpointInterface referenced from annotation symbol cannot be loaded

symbol: javax.jws.WebService

location: class com.bank.paymentmgmt.facade.BankingServiceBean

Cela signifie qu'il y a sûrement une erreur dans le nom de l'interface référencé via @WebService.endpointInterface. Assurez-vous de spécifier le bon nom pleinement qualifié de l'interface. Puis redéployez le projet Java EE bankFacade.

Si le déploiement a réussi vous devriez avoir le message suivant dans la console NetBeans pour middleware :

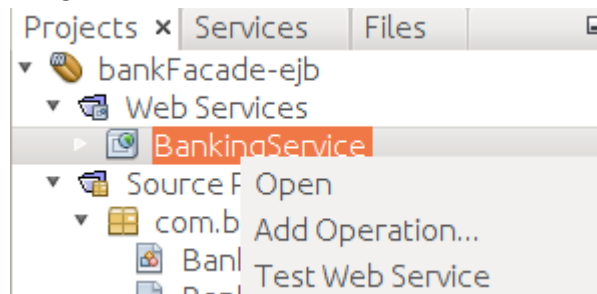
Info: EJB Endpoint deployed bankFacade-ejb

listening at address at http://prototype:11080/BankingService/BankingServiceBean

prototype est le nom de la machine utilisée pour écrire le tutoriel.

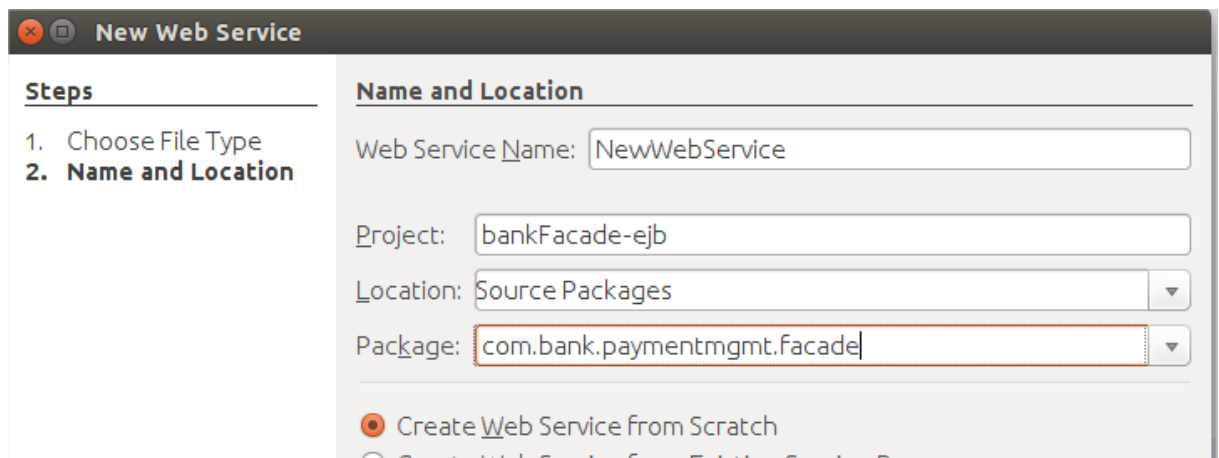
21. Testez votre web service :

clic <droite> sur BankingService puis Test Web Service pour lancer l'interface web de test du web service (<http://localhost:11080/BankingService/BankingServiceBean?Tester>). Pour cela, saisissez un numéro de compte et un montant. Prenez le temps de regarder la requête et la réponse SOAP. Testez avec un numéro de CB à 10 « chiffres » et avec un n° de CB « invalide ». Vous pouvez accéder aussi à l'interface de test, en copiant directement l'adresse dans votre navigateur.



Si le nœud Web Services n'apparaît pas dans l'onglet *Projects* de NetBeans, c'est que le service web SOAP a été déployé une première fois avec une mauvaise référence dans @WebService.endpointInterface. Même si l'erreur a été corrigée ultérieurement le nœud n'est pas créé. Vous pouvez forcer NetBeans. Pour cela :

Clic <droite> sur le module EJB > New > Sélectionnez *Web Service...* [Si non disponible, sélectionnez *Other...* > placez-vous sur la catégorie *Web Services* > et sélectionnez *Web Service*] > choisissez un packaging et cochez *Create Web Service from Scratch* > cliquez sur *Finish*.



Le nœud Web Service de l'onglet *Projects* apparaît. Ce nœud référence les 2 services Web : votre service bancaire et le nouveau service web. Vous pouvez supprimer le nouveau service web.

22. Prenez aussi le temps de regarder le WSDL généré. Le WSDL généré lors du déploiement est accessible depuis l'interface web de test – Lien *WSDL File* :

(<http://localhost:11080/BankingService/BankingServiceBean?WSDL>)

Votre point de terminaison fonctionnant, vous allez poursuivre l'implémentation de la façade du service bancaire.

23. Créez tout d'abord une énumération `PaymentStatus` représentant le statut d'un paiement. Le statut est limité à `VALIDATED` et `CANCELLED`. Vous pouvez vous appuyer sur l'assistant NetBeans, Catégorie Java.

L'énumération est localisée dans le paquetage `com.bank.paymentmgmt.domain`.

Voici le code :

```
package com.bank.paymentmgmt.domain;

public enum PaymentStatus {
    VALIDATED, CANCELLED
}
```

24. Créez la classe `Payment` qui implémente `Serializable` car elle doit pouvoir être transportée sur le réseau.

La classe est elle aussi positionnée dans le paquetage `com.bank.paymentmgmt.domain`.

```
package com.bank.paymentmgmt.domain;
import java.io.Serializable;

public class Payment implements Serializable{
```

```
private Long id;

private PaymentStatus status;

private String ccNumber;

private Double amount;

public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public PaymentStatus getStatus() {
    return status;
}

public void setStatus(PaymentStatus status) {
    this.status = status;
}

public String getCcNumber() {
    return ccNumber;
}

public void setCcNumber(String ccNumber) {
    this.ccNumber = ccNumber;
}

public Double getAmount() {
    return amount;
}

public void setAmount(Double amount) {
    this.amount = amount;
}

@Override
```

```
public String toString() {
    return "business.domain.Payment[id="+id+" ccNumber="+ ccNumber + " Amount="+ amount+"]";
}
}
```

Cette classe *Payment* représente une version pauvre du concept de domaine « ordre de paiement ». Pauvre, car un paiement est un concept beaucoup plus complexe en terme d'attributs, de relations avec d'autres objets et de méthodes. Dans notre cas, on a un objet anémique avec simplement 4 propriétés. *Payment* pourrait être considéré comme un DTO (Data Transfer Object) c'est-à-dire un objet agrégeant ensemble des données qui doivent être transférées entre couches.

id représentant l'identité d'un ordre de paiement sera assignée avec un numéro généré.

25. Afin de ne pas ajouter encore de la complexité, vous allez simuler une source de données en implémentant un bean CDI qui stockera dans une Map les ordres de paiement. Vous aurez ainsi un stockage en mémoire des ordres de paiements.

Créez un bean CDI avec les caractéristiques suivantes :

- Paquetage : **com.bank.paymentmgmt.integration**
- Classe : **MapPaymentDAO**
- Interface : **PaymentDAO**
- Scope : **Application**.

Le plus simple pour créer la structure de ce bean est de créer une classe et une interface puis d'annoter la classe. L'interface permettrait, si on le désire, de remplacer plus facilement cet implémentation utilisant un stockage en mémoire par une implémentation accédant à une source de données.

Voici le code :

Interface
<pre>package com.bank.paymentmgmt.integration; import com.bank.paymentmgmt.domain.Payment; import java.util.List; public interface PaymentDAO { //stockage d'un ordre de paiement dans une Map Payment add(Payment payment); //suppression d'un ordre de paiement Payment delete(Long id); //recherche d'un paiement en fonction de son id Payment find(Long id); //obtention d'une liste contenant les paiements créés non supprimés List<Payment> getAllStoredPayments(); }</pre>

Classe

```
package com.bank.paymentmgmt.integration;

import com.bank.paymentmgmt.domain.Payment;
import com.bank.paymentmgmt.domain.PaymentStatus;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicLong;
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class MapPaymentDAO implements PaymentDAO {

    //compteur initialisé à 1
    private AtomicLong count = new AtomicLong(1);
    private Map<Long,Payment> payments = new ConcurrentHashMap<>();

    @Override
    public Payment add(Payment payment){
        payment.setId(count.getAndIncrement()); //on génère l'id
        payment.setStatus(PaymentStatus.VALIDATED);
        //on stocke l'ordre de paiement dans la Map - la clé est l'id.
        payments.put(payment.getId(), payment);
        return payment;
    }

    @Override
    public Payment delete (Long id){
        Payment deletedPayment = payments.remove(id); //on supprime de l'entrée
        //correspondant à l'id passé
        if(deletedPayment == null){
            return null;
        }
        deletedPayment.setStatus(PaymentStatus.CANCELLED);
        return deletedPayment;
    }

    @Override
    public Payment find(Long id) {
        return payments.get(id); //récupération dans la Map de l'objet Payment
        //associé à la clé
    }

    @Override
    public List<Payment> getAllStoredPayments(){
        List<Payment> paymentList = new ArrayList<>(payments.values());
    }
}
```

```
//Boucle pour tracer la liste - pourra être supprimée par la suite
for(Payment p : paymentList){
    System.out.println(p);
}
return paymentList;
}
```

On utilise l'API `java.util.concurrent` (`AtomicLong` et `ConcurrentHashMap`) car notre bean étant accessible de manière concurrente par plusieurs clients, on doit le rendre Thread-safe. Les méthodes *delete* et *getAllStoredPayments* entreront vraiment en action dans la partie RESTful.

26. Modifiez l'implémentation `BankingServiceBean` du service web SOAP pour intégrer le DAO. Injectez le DAO et modifiez la méthode `createPayment`.

Voici l'extrait de code avec en gras les ajouts :

```
public class BankingServiceBean implements BankingServiceEndpointInterface {

    @Inject
    private PaymentDAO paymentDAO;

    @Override
    public Boolean createPayment(String ccNumber, Double amount) {
        if(ccNumber.length()== 10 ){
            System.out.println("Montant payé : "+amount+" €");
            Payment p = new Payment();
            p.setCcNumber(ccNumber);
            p.setAmount(amount);
            //pour l'instant le retour n'est pas utilisé
            p = paymentDAO.add(p);
            //juste pour tester
            paymentDAO.getAllStoredPayments();
            return true;
        } else {
            return false;
        }
    }
}
```

27. Cliquez sur *Clean and Build* puis sur *Run* pour relancer le module `bankFacade-ejb` puis testez. Si vous créez plusieurs paiements valides, vous verrez ces paiements stockés, affichez dans la console NetBeans du domaine middleware


```

Output x
Java DB Database Process x middleware x store x Retriever Output x Run (webStore) x
Info: wsimport successful
Info: Invoking wsimport with http://localhost:11080/BankingService/Bankin
Info: parsing WSDL...
Info: Generating code...
Info: Compiling code...
Info: wsimport successful
Info: Montant payé : 18.0 €
Info: business.domain.Payment[id=1 ccNumber=1231231234 Amount=12.99]
Info: business.domain.Payment[id=2 ccNumber=7894561230 Amount=18.0]
  
```

Créer l'application cliente du web service SOAP

Domaine store

28. Dans NetBeans, Créez un nouveau projet Maven de type **Web Application** :

- Nommez-le par exemple **webStore**
- Localisez-le à la racine de votre dossier projets.
- Donnez un Group Id : **com.store**

Utilisez le Group Id comme nom de paquetage par défaut.

Steps	Name and Location
1. Choose Project	Project Name: webStore
2. Name and Location	Project Location: /home/cesi/javaProjects/v2017/v1 Browse...
3. Settings	Project Folder: /home/cesi/javaProjects/v2017/v1/webStore
	Artifact Id: webStore
	Group Id: com.store
	Version: 1.0-SNAPSHOT
	Package: com.store (Optional)

- Associez-le au domaine **store**
- Sélectionnez la version Java EE 7 Web et cliquez sur Finish.

Steps

1. Choose Project
2. Name and Location
3. **Settings**

Settings

Server:

Java EE Version:

29. Comme pour le module EJB, mettez à jour le fichier pom.xml :

- Donnez un nom final à l'archive : **webStore**
- Spécifiez **Java 8** comme version utilisée.

Voici un extrait du pom.xml, avec en gras les modifications :

```
...
<build>
  <finalName>webStore</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
...
```

30. Sous le nœud *Web Pages* créez 3 pages JSF : *payment.xhtml*, *error.xhtml* et *success.xhtml* :

Clic <droite> sur le projet *webStore* ou sur le nœud *Web Pages* > New > [Other... >] * JSF Page...

* Si c'est la première fois que vous créez une Facelets, naviguez dans le menu *Other... > Catégorie JavaServer Faces*.

Assurez-vous que *Facelets* est coché et cliquez sur *Finish*.

Steps

1. Choose File Type
2. **Name and Location**

Name and Location

File Name:

Project:

Location:

Folder:

Created File:

Options:

☒ Facelets

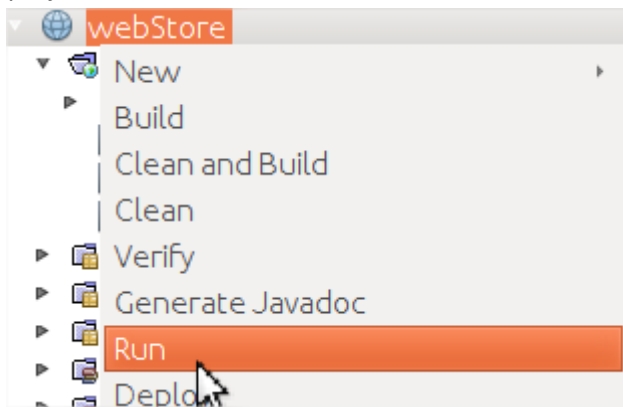
31. Dans le descripteur de déploiement *web.xml* situé sous *WEB-INF*, spécifiez la page *payment.xhtml* comme page de démarrage. Pour cela ouvrez le fichier *WEB-INF/web.xml*, et

rendez-vous en fin de fichier pour trouver l'élément <welcome-file> pointant vers index.xhtml.
effectuez le remplacement :

<welcome-file>faces /payment.xhtml</welcome-file>

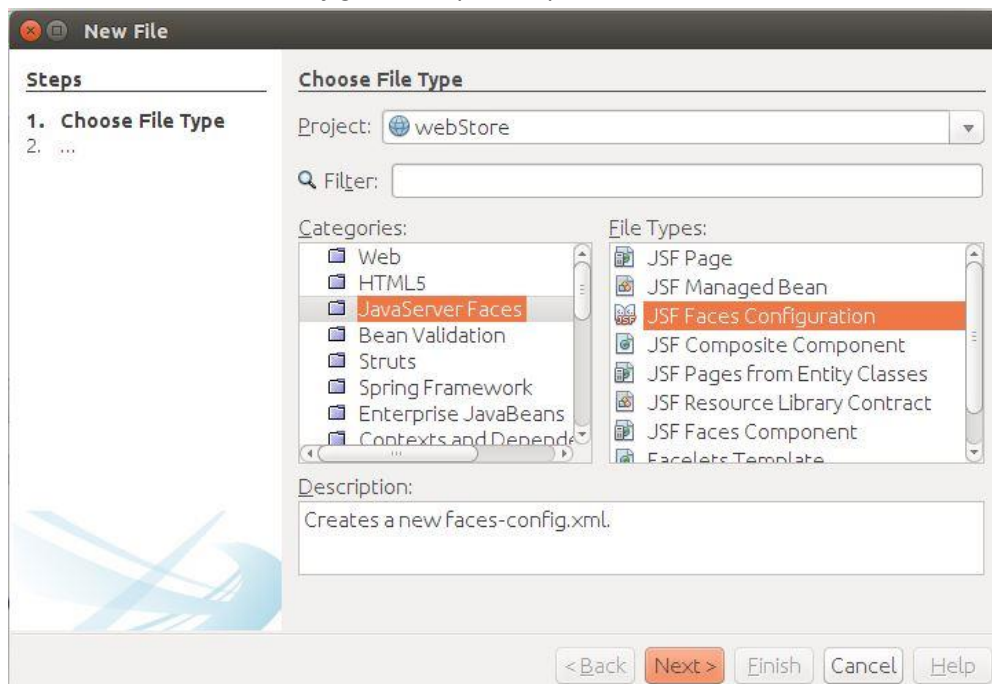
Vous pouvez supprimer la page index.html.

32. Vérifiez que votre application s'exécute correctement. Pour cela, cliquez sur *run* au niveau du projet Web Java EE **webStore**.



Une page web affichant « Hello from Facelets » s'ouvre dans le navigateur à l'adresse <http://localhost:10080/webStore/>.

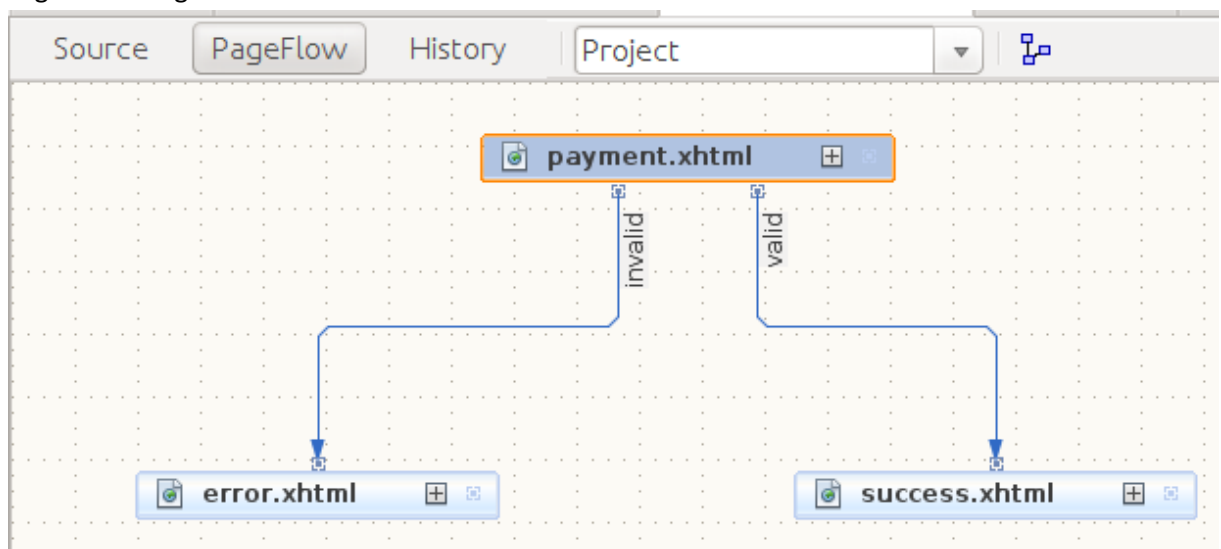
33. Ajoutez un fichier de configuration JSF pour mettre en œuvre la navigation JSF.
Ce fichier est facultatif mais permet de séparer les règles de navigation du code.
Pour cela, depuis le projet webStore, choisissez la catégorie d'éléments JavaServer Faces et sélectionnez *JSF Faces Configuration*, puis cliquez sur Next> et Finish :



34. Complétez faces-config.xml pour naviguer vers success.xhtml dans le cas de « valid » retournée. Dans le cas de « invalid » retournée, naviguez vers error.xhtml :

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.2"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">
  <navigation-rule>
    <from-view-id>payment.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>valid</from-outcome>
      <to-view-id>success.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>invalid</from-outcome>
      <to-view-id>error.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

Vous pouvez utiliser l'assistant graphique (onglet PageFlow de faces-config.xml) pour gérer les règles de navigation :



35. Créez un bean CDI (bean JSF) **PaymentBean** localisé dans le paquetage **com.store.model**. Ce bean a un scope de *requête* et est nommé **paymentModel**.
Voici comment créer ce bean en utilisant l'assistant de création de NetBeans :

Clic <droite> sur Sources Packages, puis New > [Other... > puis *Catégorie JavaServer Faces* >] Sélectionnez JSF Managed Bean > cliquez sur Next. Dans la fenêtre qui s'ouvre spécifiez le nom de la classe du Bean, le paquetage, le nom et le scope :

Steps

1. Choose File Type
2. Name and Location

Name and Location

Class Name:

Project:

Location:

Package:

Created File:

☐ Add data to configuration file

Configuration File:

Name:

Scope:

Enfin cliquez sur Finish pour que le bean soit créé.

36. Ajoutez à cette classe la propriété de type String *ccNumber* et la propriété *amount* de type Double. Créez donc 2 variables d'instance privées *ccNumber* (type String) et *amount* (type Double) ainsi que l'accesseur (getter) et le modificateur (setter) publiques pour chacune de ces variables.
37. Ajoutez au bean une méthode d'action **public String doPaymentWithSoap()**. Pour tester votre couche de présentation, Cette méthode retourne une chaine de navigation « valid » si 10 chiffres ont été saisis ou « nonvalid » dans le cas contraire.

Voici un extrait du code de votre bean CDI :

```
@Named(value = "paymentModel")
@RequestScoped
public class PaymentBean {

    private String ccNumber;
    private Double amount;

    public String doPaymentWithSoap(){
        if(ccNumber.length()==10){
            return "valid";
        }else{
            return "nonvalid";
        }
    }
}
```

```

        return "invalid";
    }
}
//getters et setters pour ccNumber et amount
....
}

```

38. Ajoutez le formulaire de saisie du paiement dans la page payment.xhtml :

```

<h:head>
    <title>Paiement</title>
</h:head>
<h:body>
    <h:form>
        <h:outputLabel value="n° carte bleue :"/>
        <h:inputText value="#{paymentModel.ccNumber}"/>
        <h:outputLabel value="montant : "/>
        <h:inputText value="#{paymentModel.amount}"/>
        <h:commandButton value="Payer avec Soap"
action="#{paymentModel.doPaymentWithSoap}"/>
    </h:form>
</h:body>

```

39. Dans la vue error.xhtml ajoutez le code suivant :

```

<h:head>
    <title>Erreur</title>
</h:head>
<h:body>
    <h1> <h:outputLabel>Erreur dans le paiement </h:outputLabel></h1>
    <br/>
    <h:form>
        <h:commandLink value="essayer de nouveau" action="payment"/>
    </h:form>
</h:body>

```

Remarquez qu'on utilise le nom de la page (payment) pour naviguer. Ici on n'utilise pas le fichier faces-config pour déclarer les règles de navigation, bien qu'on le puisse.

40. Dans la vue success.xhtml ajoutez le code suivant :

```

<h:head>
    <title>succès</title>
</h:head>
<h:body>

```

```
<h1> <h:outputLabel>paiement valide</h:outputLabel></h1>
<br/>
<h:form>
    <h:commandLink value="effectuez un nouveau paiement" action="payment"/>
</h:form>
</h:body>
```

41. Testez votre application web (*run webStore*) en saisissant un numéro de carte à 10 chiffres et aussi un numéro de carte non valide.

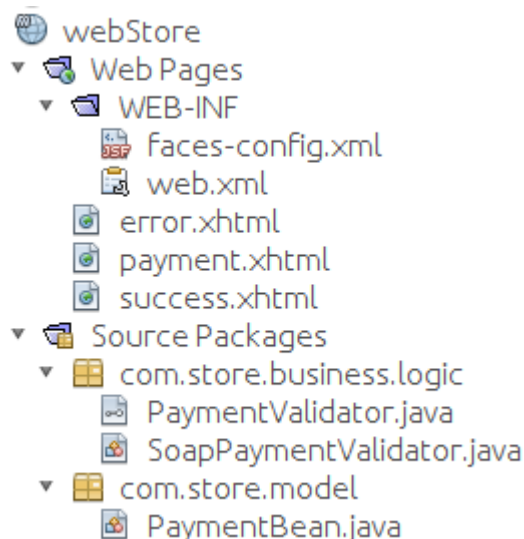
42. Dans le projet webStore, créez un Stateless Session Bean positionné dans le paquetage *com.store.business.logic*. Ce Session Bean expose une interface locale.

Le bean Stateless est nommé *SoapPaymentValidator* (c'est le nom de base de la classe) et son interface est *PaymentValidator*.

Le squelette de votre Session Bean devrait être le suivant :

Interface métier locale du Session Bean
@Local public interface PaymentValidator{}
Classe du Session Bean
@Stateless public class SoapPaymentValidator implements PaymentValidator{}

Voici une partie de l'arborescence du projet webStore :



43. Dans l'interface métier déclarez la méthode métier *public Boolean process(String ccNumber, Double amount)* et Implémentez-la dans la classe du session bean (si le bean implémente une interface).

Voici, le code de la classe du Session Bean :

```
package com.store.business.logic;
```

```
import javax.ejb.Stateless;

@Stateless
public class SoapPaymentValidator implements PaymentValidator {

    @Override
    public Boolean process(String ccNumber, Double amount) {
        Boolean isValid= true;
        return isValid;
    }
}
```

44. Injectez Le Session Bean SoapPaymentValidator dans le bean CDI PaymentBean :

```
@Inject
private PaymentValidator paymentValidator;
```

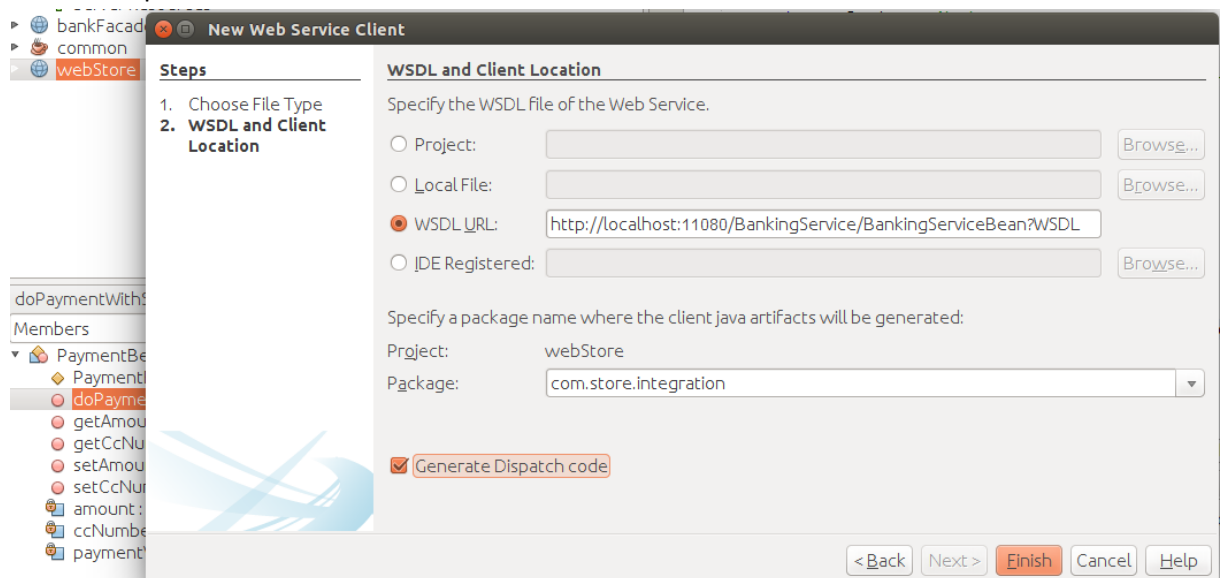
45. Modifiez la méthode doPayment du bean PaymentBean pour que le retour de la méthode PaymentBean.doPaymentWithSoap() soit fonction du retour de PaymentValidator.process(...). Voici le code modifié :

```
public String doPaymentWithSoap (){
    System.out.println("Le paiement commence");
    boolean isValid = paymentValidator.process(ccNumber, amount);

    if(isValid==true){
        return "valid";
    }else{
        return "invalid";
    }
}
```

46. Testez votre application web (*Run webStore*) en saisissant un numéro de carte à 10 chiffres ou un numéro de carte non valide. Le résultat sera toujours considéré comme valide car pour l'instant l'invocation du Session Bean retourne toujours true.
47. Assurez-vous que le domaine middleware est démarré et que le module EJB bankFacade-ejb hébergeant le web service est déployé. Si le test ne « démarre » pas, déployez bankFacade-ejb et testez le web service.
48. Dans le projet webStore, créez le client du web Service : *clic <droite> sur le projet > New > [Other... > catégorie Web Services >] Web Service Client*

- cochez WSDL URL > saisissez l'adresse du WSDL. Pour cela référez-vous à l'adresse du wsdl accessible depuis la page de test du web service. Ce devrait être : <http://localhost:11080/BankingService/BankingServiceBean?WSDL> (cf. plus haut).
- Nommez le package dans lequel les artefacts clients seront générés **com.store.integration**.
- cochez **Generate Dispatch Code**.
- Cliquez sur **Finish**.



49. Parcourez les artefacts générés (*Generated Sources (jaxws-wsimport)*). Vous trouverez parmi eux, la classe service (BankingService) et l'interface du point de terminaison BankingEndpoint.

50. Injectez dans le Session Bean SoapPaymentValidator l'interface « endpoint ».
Code pour l'injection l'interface :

```
@WebServiceRef(BankingService.class)
private BankingEndpoint banking;
```

@WebServiceRef est localisée dans le paquetage **javax.xml.ws**.

51. Dans la méthode du Session Bean *process(String ccNumber, Double amount)* invoquez la méthode de paiement sur l'interface endpoint.

Voici le code du Session Bean avec l'invocation du web service :

```
@Stateless
public class SoapPaymentValidator implements PaymentValidator{

    @WebServiceRef(BankingService.class)
    private BankingEndpoint banking;
```

```
@Override
public Boolean process(String ccNumber, Double amount) {
    Boolean isValid= banking.paymentOperation(ccNumber, amount);
    return isValid;
}
}
```

52. Testez votre application web (*run webStore*) en saisissant un numéro de carte à 10 chiffres et aussi un numéro de carte non valide. Dans la console de sortie NetBeans du domaine **middleware** hébergeant le service web, vérifiez que vous avez un message indiquant le montant payé dans le cas d'un numéro de carte bleue valide.

Partie 2 : Création de services Web RESTful et d'un client de services web

Créer un web service RESTful avec le standard JAX-RS

Domaine middleware

Vous allez maintenant créer un service web RESTful qui permettra aussi de poster un ordre de paiement mais aussi de lister les paiements créés et de les annuler.

53. Créez une application web Maven pour héberger le service REST. Cette application sera associée au domaine middleware :

- Nom : **bankFacade-war**
- Group Id : **com.bank**
- Paquetage : **com.bank.paymentmgmt.facade**

Steps	Name and Location
1. Choose Project	Project Name: <input type="text" value="bankFacade-war"/>
2. Name and Location	Project Location: <input type="text" value="/home/cesi/javaProjects/v2017/v1"/> <input data-bbox="1390 1111 1493 1144" type="button" value="Browse..."/>
3. Settings	Project Folder: <input type="text" value="/cesi/javaProjects/v2017/v1/bankFacade-war"/>
	Artifact Id: <input type="text" value="bankFacade-war"/>
	Group Id: <input type="text" value="com.bank"/>
	Version: <input type="text" value="1.0-SNAPSHOT"/>
	Package: <input type="text" value="com.bank.paymentmgmt.facade"/> (Optional)

- Associez l'application au domaine **middleware**.
- Le projet créé, modifiez comme précédemment le pom.xml pour que Java 8 soit utilisé et pour que le nom final de l'archive soit *bankFacade-war*.

Voici un extrait du pom.xml, avec en gras les modifications :

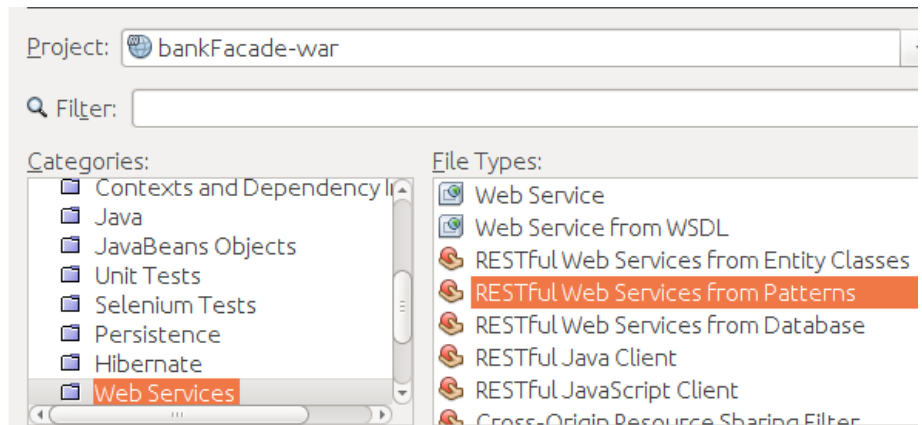
```
...
<build>
  <finalName>bankFacade-war</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.8</source>
      </configuration>
    </plugin>
  </plugins>
</build>
```

<target>1.8</target>

...

54. Pour créer le squelette du service RESTful, cliquez du <droite> sur bankFacade-war > New [> Other... > catégorie Web Services] > RESTful Web Services from Patterns

1. Choose File Type
2. ...



- Cliquez sur Next >
- Dans l'écran suivant, laissez coché *Simple Root Resource* et cliquez sur Next>
- Dans l'écran suivant vous allez configurer votre service :
 - o Spécifiez le packaging dans lequel le service va être créé : *com.bank.paymentmgmt.facade* (c'est le même packaging que pour le service SOAP)
 - o Indiquez le chemin relatif d'accès à ce service (Path) : *payment*
 - o Donnez un nom à la classe du service : *PaymentResource*
 - o Sélectionnez dans la liste déroulante MIME Type (type de média), le format des données échangées avec les clients : *application/json*
 - o Enfin cliquez sur Finish dans la fenêtre principale.

1. Choose File Type
2. Select Pattern
3. Specify Resource Classes

Project:	bankFacade-war
Location:	Source Packages
Resource Package:	com.bank.paymentmgmt.facade
Path:	payment
Class Name:	PaymentResource
MIME Type:	application/json
Representation Class:	java.lang.String Select...

Voici l'extrait des classes générées dans le packaging com.bank.paymentmgmt.facade :

Classe du service REST
<pre>@Path("payment") public class PaymentResource {</pre>

```
@Context
private UriInfo context;

...
/**
 * Retrieves representation of an instance of com.bank.paymentmgmt.facade.PaymentResource
 * @return an instance of java.lang.String
 */
@GET
@Produces(MediaType.APPLICATION_JSON)
public String getJson() {
    //TODO return proper representation object
    throw new UnsupportedOperationException();
}

/**
 * PUT method for updating or creating an instance of PaymentResource
 * @param content representation for the resource
 */
@PUT
@Consumes(MediaType.APPLICATION_JSON)
public void putJson(String content) {
}
}
```

Classe de configuration de l'application REST

```
@javax.ws.rs.ApplicationPath("webresources")
public class ApplicationConfig extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        addRestResourceClasses(resources);
        return resources;
    }

    /**
     * Do not modify addRestResourceClasses() method.
     * It is automatically populated with
     * all resources defined in the project.
     * If required, comment out calling this method in getClasses().
     */
    private void addRestResourceClasses(Set<Class<?>> resources) {
        resources.add(com.bank.paymentmgmt.facade.PaymentResource.class);
    }
}
```

Le paquetage **javax.ws.rs** correspond à l'API serveur JAX-RS.

Nous avons utilisé l'assistant NetBeans surtout pour la génération automatique de la classe *Application* qui permet de configurer une application RESTful. Cette classe annotée avec

@ApplicationPath et héritant de *javax.ws.rs.core.Application* permet la localisation des services web RESTful. Cette classe de configuration de l'application est mappée avec une Servlet générée par container Web qui interceptera toutes les requêtes HTTP préfixées **webresources**¹ pour les dispatcher au service RESTful concerné.

Vous pouvez supprimer le contenu (les méthodes) de la classe ApplicationConfig, c'est-à-dire l'enregistrement manuel des ressources REST. Dans ce cas le moteur JAX-RS détectera toute ressource annotée avec @Path.

Vous obtenez la classe « vide » suivante :

```
package com.bank.paymentmgmt.facade;

import javax.ws.rs.core.Application;

@javax.ws.rs.ApplicationPath("webresources")
public class ApplicationConfig extends Application {}
```

*Le paquetage **javax.ws.rs.core** contient les artefacts aussi bien utilisés pour développer la partie serveur que la partie cliente.*

55. @ApplicationPath permet de spécifier l'URI de base commune à l'ensemble des services RESTful de l'application. Au niveau de cette annotation, modifiez donc le *chemin* de l'application REST pour utiliser le fragment **banking** à la place de webresources.

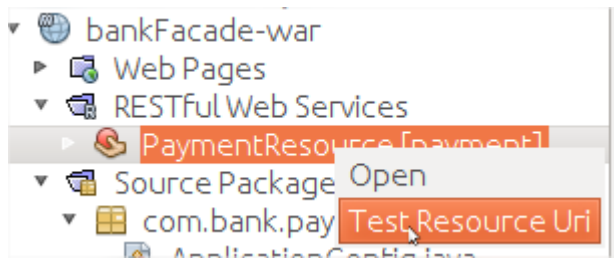
```
@javax.ws.rs.ApplicationPath("banking")
public class ApplicationConfig extends Application {}
```

56. Avant de tester votre squelette de service, modifiez la méthode déclenchée pour des requêtes HTTP GET en spécifiant un « retour Hello REST au format JSON » à la place de la levée d'exception.

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public String getJson() {
    String restMsg="{\"message\":\"hello REST\"}";
    return restMsg;
}
```

57. lancez votre projet en cliquant sur *Run*. Puis dans l'onglet Projects de NetBeans sous le noeud bankFacade-war > RESTful Web Services, cliquez du <droite> sur le service PaymentResource[payment] et sélectionnez Test Resource URI :

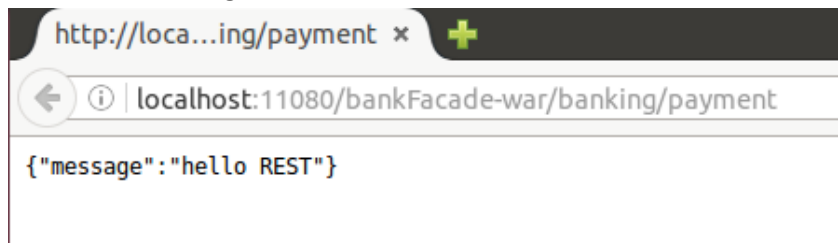
¹ Il s'agit ici de toutes les requêtes HTTP à des ressources matchant avec des URI localhost:11080/bankFacade-war/**webresources**/*



Le test exécute une requête GET à l'adresse

<http://localhost:11080/bankFacade-war/banking/payment>

Si le service est accessible à l'adresse indiquée alors le navigateur s'ouvre sur une page affichant le message.



Notez dans l'URI le chemin relatif *payment* pour accéder au service implémenté par *PaymentResource*. Ce chemin est spécifié grâce à l'annotation **@Path** sur la classe du service web RESTful.

58. Vous allez maintenant commencer à implémenter votre service RESTful pour poster des paiements. L'objectif est d'avoir une méthode déclenchée par un POST HTTP.

- Supprimez la variable d'instance *context* de type *UriInfo*
- Supprimez La méthode *getJson()* annotée avec *@GET*
- Supprimez la méthode *putJson()* annotée avec *@PUT*
- Créer une méthode publique **pay** prenant un argument *content* de type *String* et ayant comme type de retour *javax.ws.rs.core.Response*
A noter que *content* correspond au corps de la requête POST traitée par cette méthode.

```
Public Response pay(String content) {...}
```

Annotez cette méthode avec :

- *@javax.ws.rs.POST* : comme on ne connaît pas l'URI du paiement à créer on préfère invoquer le service Web au travers d'une requête POST plutôt que PUT.
- *@javax.ws.rs.Consumes(MediaType.APPLICATION_JSON)* : Les données du corps de la requête POST contenant les informations de paiement devront être au format JSON.

Retournez une réponse vide avec un code indiquant que la requête a été acceptée mais que le processus n'est pas encore complété (normal, un paiement bancaire n'est pas immédiat).

Voici l'implémentation de la méthode *pay* :

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response pay(String content) {
    //affichage du corps de la requête POST.
    System.out.println(content);
    //retour d'une réponse sans corps indiquant un statut 202 : la requête a été acceptée mais
    le processus n'est pas terminé
    return Response.accepted().build();
}
```

59. Vous pouvez tester la méthode au moyen de l'utilitaire cURL. Cet utilitaire est souvent installé nativement dans les distributions Linux, notamment dans Ubuntu. Si vous êtes sous Windows² ou sur une version Linux n'ayant pas cet utilitaire il faudra l'installer.

On considèrera que Le client doit poster un ordre de paiement au format JSON dont la structure est la suivante :

```
{
  "ccNumber": "chaîne représentant un numéro de carte",
  "amount": nombre de type double (ex 12.3 sans guillemet)
}
```

Ci-dessous, voici comment vous pouvez exécuter avec cURL une requête POST contenant un ordre de paiement :

```
curl -v -X POST -H "Content-type: application/json" -d '{"ccNumber":
"1234567891","amount": 13.5}' http://localhost:11080/bankFacade-war/banking/payment
```

Vous devriez avoir une sortie de ce type :

```
* Hostname was NOT found in DNS cache
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 11080 (#0)
> POST /bankFacade-war/banking/payment HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:11080
> Accept: text/plain
> Content-type: application/json
> Content-Length: 41
>
* upload completely sent off: 41 out of 41 bytes
< HTTP/1.1 202 Accepted
* Server Payara Server 4.1.1.161.1 #badassfish is not blacklisted
```

² Pour Windows, vous pouvez télécharger cURL à cette adresse : <https://curl.haxx.se/download.html>


```
< Server: Payara Server 4.1.1.161.1 #badassfish
< X-Powered-By: Servlet/3.1 JSP/2.3 (Payara Server 4.1.1.161.1 #badassfish Java/Oracle
Corporation/1.8)
< Date: Wed, 22 Mar 2017 10:18:48 GMT
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

La sortie ci-dessus correspond à la requête POST envoyée (lignes avec le chevron >) et à la réponse retournée par le serveur (lignes avec le chevron <).

Notez notamment le code 202 retourné avec la réponse.

60. Vous allez maintenant faire communiquer votre service REST *PaymentResource* avec le session bean *BankingServiceBean*. Ces 2 composants sont situés dans 2 modules différents. Il faut donc par conséquent exposer le session bean au travers d'une interface / vue « Remote ».

Créez donc dans le module *bankFacade-ejb* une interface Java *BankingServiceRemote* qui sera implémentée par le Session Bean :

- L'interface est localisée dans le paquetage *com.bank.paymentmgmt.facade*.
- L'interface hérite de l'interface *BankingServiceEndpointInterface*.
- L'interface est annotée avec **@javax.ejb.Remote**.
- Le session bean *BankingServiceBean* implémente aussi cette interface.

Voici un extrait du code :

Interface	
package com.bank.paymentmgmt.facade;	
import javax.ejb.Remote;	
@Remote	
public interface BankingServiceRemote extends BankingServiceEndpointInterface {}	
Classe du session bean	
@Stateless	
@WebService(
endpointInterface	=
"com.bank.paymentmgmt.facade.BankingServiceEndpointInterface",	
portName = "BankingPort",	
serviceName = "BankingService"	
)	
public class BankingServiceBean implements BankingServiceEndpointInterface,	
BankingServiceRemote {...}	

Le session bean est désormais considéré comme une façade « Dual View » car il est exposé au travers de 2 interfaces. Bien évidemment le service RESTful aurait pu accéder au session bean

via une communication SOAP. Cependant le remoting Java – Java sera plus performant qu'une communication via plateforme SOAP. De plus, l'interface `@Remote` définira par la suite des méthodes qui n'ont pas à être exposées à des clients de service SOAP.

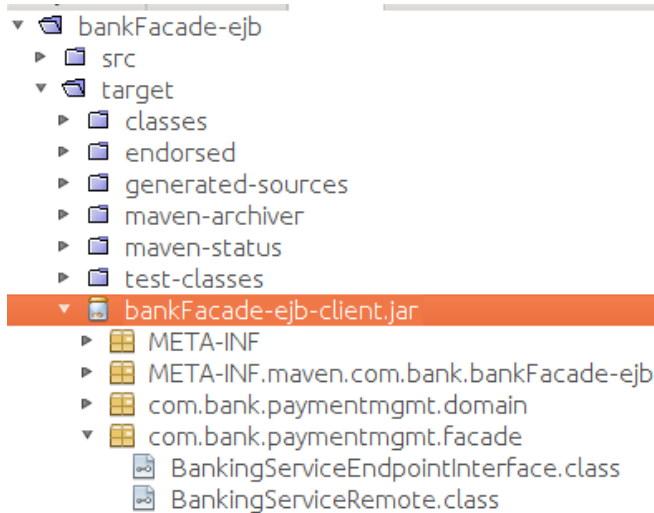
61. Le module `bankFacade-war` doit contenir une référence à la vue `@Remote` définie dans le module EJB. Vous allez donc utiliser le plugin EJB de Maven pour générer un client EJB de type jar (une bibliothèque). Ce client EJB embarquant les vues et autres artefacts devant être visibles à l'extérieur du module EJB, permettra à `bankFacade-war` de communiquer par injection avec le session bean distant.

Dans la section plugins du `pom.xml` de **bankFacade-ejb** ajoutez au plugin EJB, les informations en gras présentées ci-dessous :

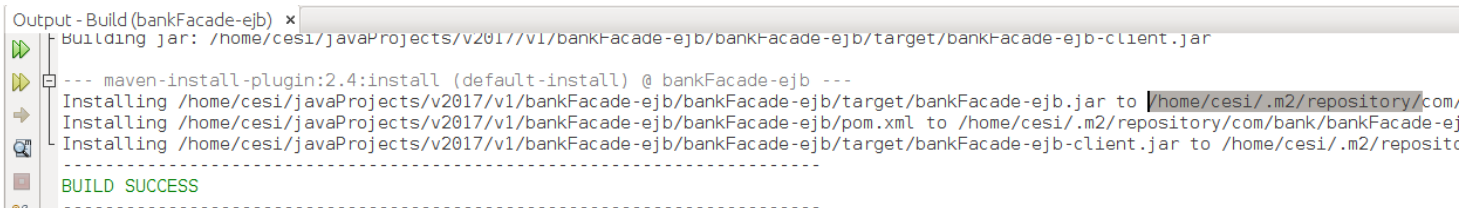
```
...
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-ejb-plugin</artifactId>
  <version>2.3</version>
  <configuration>
    <ejbVersion>3.2</ejbVersion>
    <!--génération d'un jar contenant la vue cliente -->
    <generateClient>true</generateClient>
    <clientExcludes>
      <!--exclusion de l'implémentation-->
      <clientExclude>
        com/bank/paymentmgmt/facade/BankingServiceBean.class
      </clientExclude>
      <!--exclusion du DAO-->
      <clientExclude>
        com/bank/paymentmgmt/**/integration/
      </clientExclude>
    </clientExcludes>
  </configuration>
</plugin>
...
```

L'application `bankFacade-war` n'a pas besoin de connaître l'implémentation du session bean. C'est pour que la classe est exclue. On exclut aussi le DAO qui n'a pas à être visible à l'extérieur du module EJB.

62. Lancez un Clean and Build de `bankFacade-ejb` pour générer le jar `bankFacade-ejb-client.jar`. le jar client généré est visible dans l'onglet Files de NetBeans >`bankFacade-ejb` > Target. Si vous déroulez le jar vous verrez quels sont les artefacts embarqués.



Dans la console de sortie NetBeans pensez à regarder où est situé le dépôt local Maven. Vous en aurez besoin dans l'étape suivante pour ajouter au module une dépendance au client EJB



Exécutez *Run* pour redéployer bankFacade-ejb.

63. Ajoutez dans le pom.xml de *bankFacade-war* une dépendance au client EJB. Il faut au préalable référencer dans ce pom.xml, le dépôt local Maven.

En gras, les modifications à apporter au pom.xml de bankFacade-war :

```
...
<repositories>
  <repository>
    <id>localrepo</id>
    <url>file:///home/cesi/.m2/repository</url>
  </repository>
</repositories>

<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>bankFacade-ejb</artifactId>
    <version>${project.version}</version>
    <type>ejb-client</type>
  </dependency>
  <dependency>
    <groupId>javax</groupId>
```

```
<artifactId>javaee-web-api</artifactId>
<version>7.0</version>
<scope>provided</scope>
</dependency>
</dependencies>
...
```

/home/cesi/.m2/repository est l'emplacement sous Linux du dépôt local Maven pour un utilisateur nommé **cesi**. Pour Windows ce serait *C:\Users\cesi\.m2\repository*.

Le pom.xml modifié effectuez un **Clean & Build** (ou Build with Dependencies) sur *bankFacade-war* pour régénérer l'application REST avec la nouvelle dépendance.

64. Annotez la classe `PaymentResource` avec `@RequestScoped` pour activer le support de l'injection dans ce composant. Notez que telle que l'application REST est configurée, Le moteur JAX-RS crée par défaut une instance de service par requête. Par conséquent l'annotation `@RequestScoped` ne change pas le cycle de vie par défaut du service REST.

65. Injectez dans `PaymentResource` (*bankFacade-war*) la vue « remote » du Session Bean `BankingServiceBean` (*bankFacade-ejb*) en spécifiant le nom portable de celui-ci :

```
@EJB(lookup = "java:global/bankFacade-ejb/BankingServiceBean")
private BankingServiceRemote bankingService;
```

Le nom portable JNDI³ de l'EJB `BankingServiceBean` est affiché dans la console de sortie NetBeans du domaine `middleware` quand le module *bankFacade-ejb* est déployé. Ci-dessous un extrait des logs de `middleware` indiquant les 2 noms portables associés à `BankingServiceBean`.

```
Info: Portable JNDI names for EJB BankingServiceBean:
[java:global/bankFacade-ejb/BankingServiceBean, java:global/bankFacade-ejb/BankingServiceBean!com.bank.paymentmgmt.facade.BankingServiceRemote]
```

On utilise l'annotation `@EJB` pour injecter une vue remote car l'annotation `@Inject` liée à la spécification CDI ne supporte pas le `remoting`. L'attribut `lookup` permet de spécifier le nom global standard sous lequel l'EJB est enregistré dans l'annuaire du serveur d'application lors du déploiement. Ce nom standard permet au container de rechercher un composant qui n'est pas situé dans la même application. Il faut cependant que l'EJB soit déployé dans le même domaine. Si ce n'était pas le cas, il faudrait utiliser un mécanisme propriétaire du serveur d'application.

66. Modifiez la méthode `pay` de `PaymentResource` pour qu'elle utilise le session bean `BankingServiceBean` afin de valider l'ordre de paiement.

³ JNDI: Java Naming and Directory Interface. API fournissant un service d'annuaires et de nommages des composants Java.

- Il faut tout d'abord extraire du corps de la requête HTTP, le numéro de carte et le montant payé. Pour rappel, le paramètre *content* (type String) de **pay** correspond à ce corps de requête. Pour cela vous allez utiliser le standard Java EE JSON-P (Java API for JSON Processing) relatif à la manipulation de données au format JSON. Voici le code :

```
StringReader reader = new StringReader(content);
String ccNumber;
Double amount;
try (JsonReader jreader = Json.createReader(reader)) {
    JsonObject paymentInfo = jreader.readObject();
    ccNumber = paymentInfo.getString("ccNumber");
    amount = paymentInfo.getJsonNumber("amount").doubleValue();
}
```

*StringReader appartient au paquetage **java.io** et les types JSON-P appartiennent tous au paquetage **javax.json**.*

- Ensuite il faut invoquer la méthode `createPayment` du Session Bean en lui passant en paramètre le numéro de carte bleue et le montant précédemment récupérés :

```
Boolean isValid = bankingService.createPayment(ccNumber, amount);
```

- Enfin, en fonction de la valeur booléenne retournée par le Session Bean, il faut créer l'objet **Response** à retourner. Si la valeur est *true*, alors on retourne une réponse avec un code HTTP 202 signifiant que la requête a été acceptée mais que le traitement n'est pas encore finalisé (comme vu précédemment). Dans le cas contraire, la réponse retournée correspond à une erreur client (code 400 : bad Request). Dans ce dernier cas, nous spécifions un message décrivant l'erreur.

Voici le code :

```
Response resp = null;
if(isValid){
    resp = Response.accepted().build();
}else{
    resp = Response.status(400).entity("n° CB invalide").build();
}
return resp;
```

Voici donc l'implémentation finale `PaymentResource` avec les points clés en gras :

```
package com.bank.paymentmgmt.facade;

import java.io.StringReader;
import javax.ejb.EJB;
import javax.enterprise.context.RequestScoped;
import javax.json.*;
```

```
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;

import javax.ws.rs.Path;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

/**
 * REST Web Service
 *
 * @author cesi
 */
@Path("payment")
@RequestScoped
public class PaymentResource {

    @EJB(lookup = "java:global/bankFacade-ejb/BankingServiceBean")
    private BankingServiceRemote bankingService;

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response pay(String content) {
        //affichage du corps de la requête POST.
        System.out.println(content);

        StringReader reader = new StringReader(content);
        String ccNumber;
        Double amount;
        try (JsonReader jreader = Json.createReader(reader)) {
            JsonObject paymentInfo = jreader.readObject();
            ccNumber = paymentInfo.getString("ccNumber");
            amount = paymentInfo.getJsonNumber("amount").doubleValue();
        }

        Boolean isValid = bankingService.createPayment(ccNumber, amount);

        Response resp = null;
        if(isValid){
            resp = Response.accepted().build();
        }else{
            resp = Response.status(400).entity("n° CB invalide").build();
        }
    }
}
```

```

    }
    return resp;
  }
}

```

Vous pouvez supprimer le `System.out.println(content)`.

67. Reconstituez le module `bankFacade-war` (Clean & Build) et redéployez-le (Run). Profitez-en pour vérifier que `bankFacade-ejb` est bien déployé sur le serveur.

Testez avec `cURL` comme précédemment en postant un paiement avec un numéro de carte bleue valide (10 « chiffres ») et aussi un paiement avec un numéro non valide.

Dans la console NetBeans du domaine middleware, vous pouvez voir aussi la « liste » des paiements s'incrémenter à chaque nouveau paiement validé.

Il faut maintenant implémenter la récupération des ordres de paiement stockés, la déletion et la recherche d'un paiement spécifique puis tester tout ça avec `cURL`.

68. Vous allez utiliser les annotations JAX-B afin d'indiquer au moteur JAX-RS comment convertir un objet **Payment** en message XML ou JSON.

Cela permettra de mettre en œuvre le principe de négociation de contenu (Content Negotiation –Conneg). Il s'agit de la capacité à fournir différentes représentations du message contenu dans le corps de la réponse http en fonction de type de réponse attendu par le client. Ici on se limite à des corps d'entité au format XML et JSON.

Le standard JAX-B (Java Architecture for XML Binding) est un standard initialement destiné à mapper des objets Java avec des structures XML, un peu à la manière de JPA qui permet, entre autre, de mettre en correspondance un modèle objet et un modèle relationnel.

La plupart des moteurs JAX-RS du marché utilisent des bibliothèques intégrées telle que jettison pour fournir la capacité de convertir un objet Java en JSON et vice-versa. Cette capacité n'est pas un standard Java EE⁴.

Voici les extraits de l'énumération `PaymentStatus` et de la classe `Payment` annotées :

```

import javax.xml.bind.annotation.XmlEnum;

@XmlEnum
public enum PaymentStatus {
    VALIDATED, CANCELLED
}

...
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Payment implements Serializable{
    @XmlAttribute
    private Long id;
}

```

⁴ Java EE 8 intégrera le standard JSON-B pour le mapping JSON/Java.

```
@XmlElement
private PaymentStatus status;

@XmlElement
private String ccNumber;

@XmlElement
private Double amount;
//getters et setters
}
```

Les annotations JAX-B appartiennent au paquetage **javax.xml.bind.annotation**.

@XmlElement sert à mapper une propriété Java avec un élément XML. **@XmlAttribute** permet de mettre en correspondance une propriété java avec un attribut d'élément XML.

69. Déclarez dans l'interface **BankingServiceRemote** annotée **@Remote** (module **bankFacade-EJB**) les méthodes :

- *List<Payment> lookupAllStoredPayments()* : récupère tous les paiement stockés dans la Map du DAO.
- *Payment lookupStoredPayment(Long id)* : récupère un paiement en fonction de son identité.
- *Payment deleteStoredPayment(Long id)* : supprime de la Map un paiement.

Implémentez ensuite ces méthodes dans la classe du bean **BankingServiceBean**. Ces méthodes utilisent le DAO pour rechercher et supprimer.

Voici un extrait de la classe **BankingServiceBean** présentant les 3 méthodes implémentées :

```
...
@Stateless
@WebService(
    endpointInterface = "com.bank.paymentmgmt.facade.BankingServiceEndpointInterface",
    portName = "BankingPort",
    serviceName = "BankingService"
)
public class BankingServiceBean implements BankingServiceEndpointInterface,
BankingServiceRemote {

    @Inject
    private PaymentDAO paymentDAO;
    ...
    //méthodes déclarées dans BankingServiceRemote

    @Override
    public List<Payment> lookupAllStoredPayments() {
```



```

    return paymentDAO.getAllStoredPayments();
}

@Override
public Payment lookupStoredPayment(Long id) {
    return paymentDAO.find(id);
}

@Override
public Payment deleteStoredPayment(Long id) {
    return paymentDAO.delete(id);
}
}

```

70. Afin de régénérer le client EJB exposant ces méthodes et d'utiliser celui-ci dans *bankFacade-war* :

- Effectuez un build du projet *bankFacade-ejb* et redéployez-le.
- Effectuez un build de *bankFacade-war* et redéployez-le.

Vous pouvez désormais invoquer ces méthodes distantes dans votre application RESTful *bankFacade-war*.

71. Dans *PaymentRessource* (localisé dans *bankFacade-war*), créez la méthode *public Response getStoredPayments()* mappée avec une requête GET permettant de retourner la liste des paiements stockés. Cette méthode doit pouvoir produire un contenu de réponse au format *application/json* ou *application/xml*. Cette méthode sera invoquée pour toute requête http GET à

<http://localhost:11080/bankFacade-war/banking/payment/payments>

Voici l'implémentation de la méthode *PaymentResource.getStoredPayments* :

```

@Path("payments")
@GET
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public Response getStoredPayments(){
    //récupération de tous les ordres de paiement stockés
    List<Payment> storedPayments = bankingService.lookupAllStoredPayments();
    //création d'une entité générique pour pouvoir mapper un type paramétré (List<Payment>) avec
    //un corps de réponse
    GenericEntity<List<Payment>> genericList = new
        GenericEntity<List<Payment>>(storedPayments){};
    //construction de la réponse embarquant dans son corps les ordres de paiements
    Response resp = Response.ok(genericList).build();
    return resp;
}

```

On utilise `javax.ws.rs.core.GenericEntity` pour éviter le problème de Type Erasure. Le paramètre `Payment` de `List<Payment>` est supprimé par le compilateur Java lors de la génération du byte code. Par conséquent si on passait directement la liste de type `List<Payment>`, le `MessageBodyWriter` JAX-RS chargé de conversion Java en corps de message échouerait du fait de l'absence du type lors de l'exécution. JAX-RS fournit donc `GenericEntity` pour gérer ce problème.

72. Testez la récupération de la liste avec cURL. Créez au préalable des paiements comme vous l'avez fait en étape 59.

- Pour obtenir une représentation XML de la liste des paiements, utiliser l'invocation suivante :

```
cesi@prototype:~$ curl -v -X GET -H "Accept: application/xml"
http://localhost:11080/bankFacade-war/banking/payment/payments
```

Si l'implémentation est correcte le corps de la réponse contient l'ensemble des paiements postés au format XML :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><payments><payment
id="1"><status>VALIDATED</status><ccNumber>1234567191</ccNumber><amount>13.5
</amount></payment><payment
id="2"><status>VALIDATED</status><ccNumber>4569917855</ccNumber><amount>130.
0</amount></payment><payment
id="3"><status>VALIDATED</status><ccNumber>4569917855</ccNumber><amount>29.9
9</amount></payment></payments>
```

- Testez l'obtention d'une représentation JSON :

```
curl -v -X GET -H "Accept: application/json"
http://localhost:11080/bankFacade-war/banking/payment/payments
```

Si l'implémentation est correcte le corps de la réponse contient l'ensemble des paiements postés au format XML :

```
[{"id":1,"status":"VALIDATED","ccNumber":"1234567191","amount":13.5}, {"id":2
,"status":"VALIDATED","ccNumber":"4569917855","amount":130.0}, {"id":3,"statu
s":"VALIDATED","ccNumber":"4569917855","amount":29.99}]
```

73. Toujours dans `PaymentRessource` créez la méthode `public Response getStoredPayment(@PathParam("id") Long paymentId)` mappée avec une requête GET permettant un paiement en fonction de l'id spécifié dans l'URI. Cette méthode doit pouvoir produire un contenu de réponse au format `application/json` ou `application/xml`. Cette méthode sera invoquée pour toute requête http GET dont le pattern d'URI correspond à :

<http://localhost:11080/bankFacade-war/banking/payment/{id}>

Si l'id passé dans l'URL ne correspond à aucun ordre de paiement stocké dans le DAO, alors une exception JAX-RS `NotFoundException` est levée.

Voici la méthode :

```
@Path("{id}")//utilisation d'un template parameter dans le pattern d'URI
@GET
@Produces({MediaType.APPLICATION_XML,MediaType.APPLICATION_JSON})
```

```
public Response getStoredPayment(@PathParam("id") Long paymentId){//@PathParam
permet //d'extraire la valeur du template parameter
    Payment storedPayment = bankingService.lookupStoredPayment(paymentId);
    if(storedPayment==null){//si aucun ordre de paiement correspondant à l'id n'est stocké
        throw new NotFoundException();//exception mappée avec un code d'erreur 404
    }
    return Response.ok(storedPayment).build();
}
```

La valeur du template parameter {id} spécifié dans @Path est assigné au paramètre paymentId annoté avec @javax.ws.rs.PathParam.

Le moteur JAX-RS « transforme » l'exception en code d'erreur client 404.

74. Testez avec cURL l'obtention d'une représentation JSON et XML d'un ordre de paiement :

- Obtention d'un contenu au format JSON :

```
curl -v -X GET -H "Accept: application/json"
http://localhost:11080/bankFacade-war/banking/payment/1
```

- Obtention d'une représentation au format XML :

```
curl -v -X GET -H "Accept: application/xml"
http://localhost:11080/bankFacade-war/banking/payment/1
```

Vous pouvez aussi Tester aussi le retour d'erreur 404 en exécutant une requête avec un id un supérieur au nombre de paiements stockés.

75. Enfin implémentez dans votre service RESTful la déletion d'un ordre de paiement *public Response cancelStoredPayment(@PathParam("id") Long paymentId)*. La méthode est mappée avec une requête DELETE pour le pattern d'URI :

<http://localhost:11080/bankFacade-war/banking/payment/{id}>

C'est le même pattern que pour la méthode précédente de recherche d'un ordre de paiement. En cas de succès de la déletion, une réponse sans contenu indiquant un succès de l'opération est retournée (code 204) au client. Si l'id ne correspond à aucun ordre de paiement alors le code 404 d'erreur est retourné.

Voici l'implémentation de la méthode :

```
@Path("{id}")
@DELETE
public Response cancelStoredPayment(@PathParam("id") Long paymentId){
    Payment cancelledPayment = bankingService.deleteStoredPayment(paymentId);
    if(cancelledPayment==null){//si aucun ordre de paiement correspondant à l'id n'est stocké
        throw new NotFoundException();//exception mappée avec un code d'erreur 404
    }
    return Response.noContent().build();//réponse vide indiquant un succès de l'opération
}
```

Testez la suppression avec cURL :

```
curl -v -X DELETE -H "Accept: application/xml"  
http://localhost:11080/bankFacade-war/banking/payment/1
```

Si le paiement 1 a été supprimé alors la réponse contiendra un code 204. Si le paiement 1 n'est pas stocké au niveau de PaymentDAO, alors la requête retournera le code 404.

Vous pouvez tester que la requête DELETE a bien supprimé la ressource en exécutant ensuite une requête listant les paiements stockés.

Créer un client pour le service web RESTful

Domaine store

Vous allez maintenant créer dans webStore un client pour le service RESTful en utilisant l'API cliente.

Normalement les invocations REST listant les paiements ou permettant la suppression devraient être réalisées depuis une application tierce de gestion des paiements. Pour la simplicité du prototype tout sera implémenté au sein de webStore.

76. Dans le projet *webStore* créez dans le paquetage **com.store.business.logic** un Session Bean Stateless chargé d'invoquer le service RESTful. Ce Session est nommé **RestPaymentValidator** et implémente l'interface métier *PaymentValidator*, interface implémentée aussi par le Session Bean *SoapPaymentValidator* chargé d'invoquer le service web SOAP.

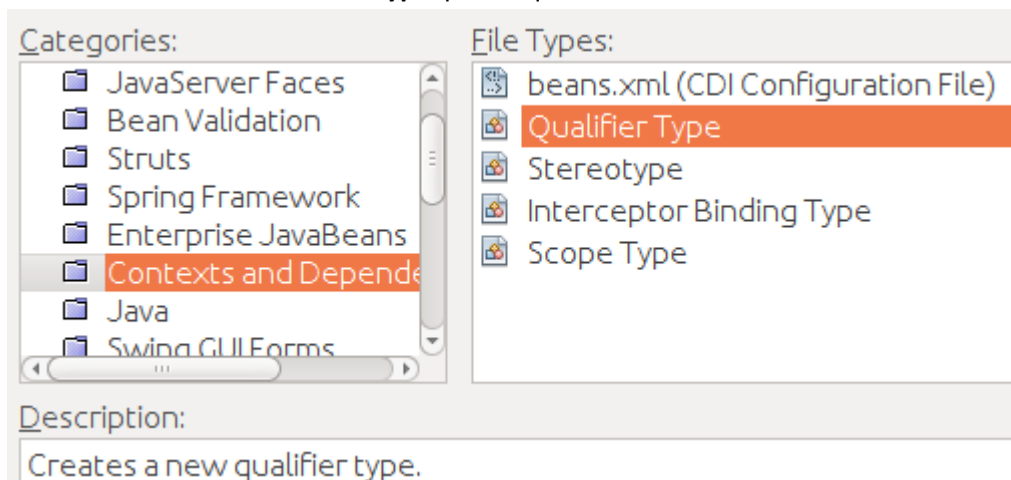
Le plus simple pour créer ce Session Bean est d'utiliser l'assistant NetBeans sans cocher la création d'une interface locale. Une fois ce bean sans interface créé vous spécifiez explicitement l'implémentation de l'interface *PaymentValidator*.

Voici le squelette du Session Bean *RestPaymentValidator* après avoir remplacé la levée d'exception par **return true** ;

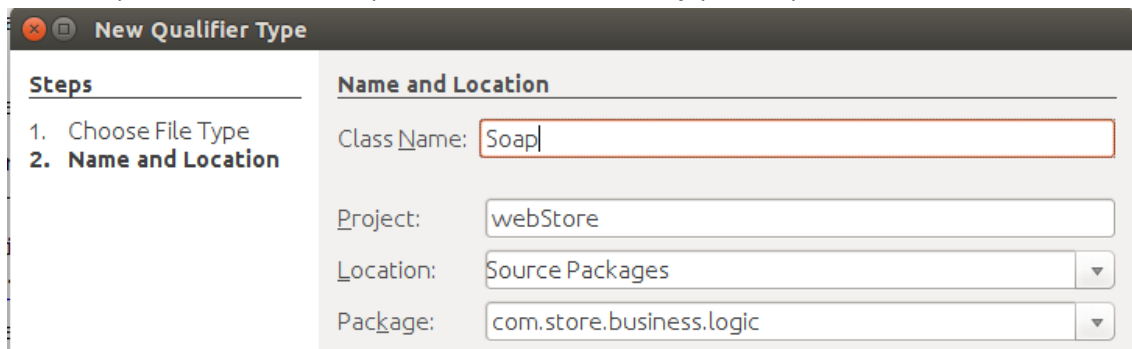
```
package com.store.business.logic;  
  
import javax.ejb.Stateless;  
  
@Stateless  
public class RestPaymentValidator implements PaymentValidator{  
  
    @Override  
    public Boolean process(String ccNumber, Double amount) {  
        return true;  
    }  
}
```

77. Vous allez créer des qualificateurs CDI pour distinguer les 2 Session Beans `PaymentValidator` et `RestPaymentValidator` qui implémentent la même interface. En effet comme il y a deux Session Beans implémentant la même interface métier, il faut pouvoir les distinguer pour préciser quel Session Bean injecté au point d'injection de type `PaymentValidator`.

- Créez le qualificateur `@com.store.business.logic.Soap`. Vous pouvez utiliser l'assistant NetBeans : clic <droite> sur le paquetage **`com.store.business.logic`**, puis sélectionnez le menu New. Si Qualifier Type n'est pas dans le menu, sélectionnez Other... Dans la fenêtre positionnez-vous sur la catégorie *Contexts And Dependency Injection* et sélectionnez **Qualifier Type** puis cliquez sur Next>



- Spécifiez dans le champ *Class Name*, le nom **`Soap`** puis cliquez sur Finish.



Voici l'extrait de code de l'annotation générée :

```
package com.store.business.logic;
//....
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Soap {
}
```

- De la même façon, toujours dans le paquetage *com.store.business.logic*, créez le qualificateur `@Rest` dont voici l'extrait :

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Rest {
}
```

78. Annotez les classes des Session Beans *SoapPaymentValidator* (client du service SOAP) et *RestPaymentValidator* (client du service RESTful) avec Respectivement `@Soap` et `@Rest` :

```
@Soap
@Stateless
public class PaymentValidator implements PaymentValidator{...}

@Rest
@Stateless
public class RestPaymentValidator implements PaymentValidator{...}
```

Vous noterez au passage qu'on peut utiliser les qualificateurs CDI avec les session beans.

79. Dans le bean CDI *com.store.model.PaymentBean*, ajoutez sur le point d'injection **paymentValidator** le qualificateur `@Soap` car cette variable référence le Session Bean invoquant le service web SOAP :

```
@Inject @Soap
private PaymentValidator paymentValidator;
```

80. Toujours dans *PaymentBean* créez un nouveau point d'injection de type **PaymentValidator**, nommé **restPaymentValidator**, dans lequel sera injecté le Session Bean Stateless *RestPaymentValidator*. Il faut donc spécifier sur cette variable d'instance le qualificateur `@Rest` :

```
@Inject @Rest
private PaymentValidator restPaymentValidator;
```

81. Dans ce Bean *paymentBean*, créez la méthode d'action **doPaymentWithRest** retournant le type **String** et ne prenant pas d'argument. L'implémentation de cette méthode est quasiment une copie conforme de l'autre méthode d'action *doPaymentWithSoap*, à la seule différence que *doPaymentWithRest* invoque le Session Bean *RestPaymentValidator*. Il faudrait donc de refactoriser ce code. Je vous laisse cette refactorisation comme exercice.

Ci-après, un extrait de l'implémentation mise à jour de *PaymentBean* :

```
package com.store.model;

import com.store.business.logic.PaymentValidatorLocal;
import com.store.business.logic.Rest;
import com.store.business.logic.Soap;
```

```
import javax.inject.Named;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;

@Named(value = "paymentModel")
@RequestScoped
public class PaymentBean {

    private String ccNumber;
    private Double amount;

    @Inject @Soap
    private PaymentValidatorLocal paymentValidator;

    @Inject @Rest
    private PaymentValidatorLocal restPaymentValidator;

    public String doPaymentWithSoap(){
        System.out.println("Le paiement commence");
        boolean isValid = paymentValidator.process(ccNumber, amount);
        if(isValid==true){
            return "valid";
        }else{
            return "invalid";
        }
    }

    public String doPaymentWithRest(){
        System.out.println("Le paiement commence");

        boolean isValid = restPaymentValidator.process(ccNumber, amount);

        if(isValid==true){

            return "valid";
        }else{
            return "invalid";
        }
    }
    ...
}
```

82. Dans la vue *payment.xhtml*, ajoutez un bouton pour effectuer un paiement via un web service RESTful – pour cela il faut que l'attribut **action** de la nouvelle balise **commandButton** « pointe vers la méthode **doPaymentWithRest** de **PaymentModel** (le nom explicite du bean CDI implémenté par la classe *PaymentBean*) :

```
<h:form>
    <h:outputLabel value="n° carte bleue :"/>
    <h:inputText value="#{paymentModel.ccNumber}"/>
    <h:outputLabel value="montant :"/>
    <h:inputText value="#{paymentModel.amount}"/>
    <h:commandButton value="Payer avec Soap"
action="#{paymentModel.doPaymentWithSoap}"/>
    <h:commandButton value="Payer avec Rest"
action="#{paymentModel.doPaymentWithRest}"/>
</h:form>
```

83. Effectuez une reconstruction de *webStore* (*Clean & Build*) puis déployez l'archive avec *Run*. Testez votre application *webStore* pour vérifier qu'à cette étape tout est correctement implémenté. Que le numéro de CB soit valide ou pas, le test du paiement via service RESTful entrainera obligatoirement l'affichage de la vue indiquant un paiement réussi car *RestPaymentValidator.process* retourne (pour l'instant) toujours *true*.

84. Vous allez utiliser l'API cliente fourni par JAX-RS (paquetage *javax.ws.rs.client*) pour invoquer votre service RESTful depuis le Session Bean *RestPaymentValidator*.

- Dans la méthode *RestPaymentValidator.process*, utilisez le model objet fourni par JSON-P pour créer un message json représentant l'ordre de paiement :

```
//création d'un builder pour créer un objet java représentant un message json
JsonObjectBuilder paymentBuilder = Json.createObjectBuilder();
//utilisation du builder pour créer la représentation JSON du paiement
JsonObject paymentObject = paymentBuilder.add("ccNumber",ccNumber)
.add("amount",amount).build();
```

*L'ensemble des artefacts est localisé dans *javax.json*.*

- Il ne reste plus qu'à invoquer le service RESTful :
 - Il faut d'abord créer grâce à un builder de l'API cliente JAX-RS, une abstraction du concept de client de service RESTful.
 - Ce client vous permet d'obtenir une référence à votre ressource REST cible. Pour cela vous invoquez sur ce client la méthode **target()** en lui passant l'adresse de la ressource REST.
 - L'étape suivante consiste à invoquer le service RESTful au moyen d'une requête *POST* contenant l'ordre paiement au format JSON et à récupérer la *réponse* du service. La méthode **request()** permet d'initier la création d'un objet représentant la requête http invocant le service. la méthode **post()**

chainée va «générée » la requête. Elle prend en paramètre l'entité, c'est-à-dire le message contenu dans le corps de cette requête. Cette entité prend en arguments l'ordre de paiement et le format dans lequel il doit être posté. Dans notre cas le format (type de média) consommé par le service REST est « application/json »).

- La requête POST exécutée, il faut fermer le client et la réponse pour libérer les ressources utilisées en coulisse par JAX-RS pour poster le paiement.
- Si le statut de la *réponse* retournée est **202** (paiement accepté) alors vous retourner **true** sinon vous retournez **false** et lisez le corps de la réponse contenant le message d'erreur.

Voici un extrait de l'implémentation avec en gras la mise en œuvre de la requête :

```
...
@Stateless
@Rest
public class RestPaymentValidator implements PaymentValidator{

    @Override
    public Boolean process(String ccNumber, Double amount) {

        JsonObjectBuilder paymentBuilder = Json.createObjectBuilder();
        JsonObject paymentObject = paymentBuilder.add("ccNumber",ccNumber)
            .add("amount",amount).build();

        Client client = ClientBuilder.newClient();
        WebTarget target = client.target("http://localhost:11080/bankFacade-war/banking/payment");
        Response resp = target.request()
            .post(Entity.entity(paymentObject.toString(), MediaType.APPLICATION_JSON_TYPE));
        boolean success;
        if(resp.getStatus()==202){
            success = true;
        }else{
            success = false;
        }
        resp.close();
        client.close();
        return success;
    }
}
```

Hormis Response et MediaType qui appartiennent à javax.rs.core, ClientBuilder, Client et WebTarget appartiennent à javax.rs.client.

85. Après avoir construit et redéployer le webStore, testez depuis l'interface web la création d'un ordre de paiement via service web RESTful.

Vous pouvez vérifier avec cURL que l'ordre de paiement a bien été stocké.

Il faut maintenant implémenter les fonctionnalités clientes permettant de lister les ordres de paiements et de les supprimer. Pour rappel, « dans la vraie vie », ces fonctionnalités ne seraient sûrement pas exposées à des magasins en ligne mais plutôt destinées à des applications tierces de gestion des paiements bancaires.

86. Créez tout d'abord une classe PaymentOrder représentant une version simplifiée du concept d'ordre de paiement. Cette classe appartient au paquetage com.store.model.

Cette classe définit 3 propriétés en lecture seule (pas de setters) :

- location de type String : adresse (URI) de l'ordre de paiement
- amount de type Double : montant payé
- orderNumber : le numéro de la commande qui est l'id généré du paiement

Créez un constructeur permettant d'assigner ces 3 propriétés.

Ci-dessous la classe que vous devriez obtenir :

```
package com.store.model;

public class PaymentOrder {

    private final String location;
    private final Double amount;
    private final Long orderNumber;

    public PaymentOrder (String location, Double amount, Long orderNumber) {
        this.location = location;
        this.amount = amount;
        this.orderNumber = orderNumber;
    }
    //getters non présentés
    ...
}
```

87. Dans com.store.model créez le bean CDI PaymentOrderBean ayant les caractéristiques suivantes :

- Scope : Requête
- Nom : paymentOrderModel

Créez la variable d'instance finale *baseUrl* de type String représentant l'URL de base du service REST.

Déclarez dans ce bean la propriété *orders* de type `List<Order>` en lecture seule (getter). *orders* est implémentée par une `ArrayList`.

Vous devriez avoir le squelette suivant :

```
@Named(value = "paymentOrderModel")
@RequestScoped
public class PaymentOrderBean {

    private final String baseUrl = "http://localhost:11080/bankFacade-war/banking/payment";
    private List<PaymentOrder> paymentOrders = new ArrayList<>();

    public List<PaymentOrder> getPaymentOrders() {
        return paymentOrders;
    }
}
```

baseUrl est l'URI de la ressource racine *PaymentResource*.

Dans ce bean implémentez la méthode privée `void loadAllPayments()` permettant de charger l'ensemble des ordres de paiements stockés au niveau du serveur REST :

- Il faut positionner en tout début de méthode l'instruction `paymentOrders.clear()` pour avoir une liste vierge ne contenant pas d'anciennes entrées.
- Pour charger la liste des paiements, il faut d'abord exécuter la requête `http://localhost:11080/bankFacade-war/banking/payment/payments` pour charger la liste des paiements :

```
WebTarget target = client.target(baseUrl).path("payments");
Response resp = target.request().accept(MediaType.APPLICATION_JSON_TYPE).get();
String jsonContent = resp.readEntity(String.class);
resp.close();
client.close();
```

`path("payments")` permet de créer une adresse cible avec le segment *payments* ajouté à l'adresse de base.

`Accept()` permet de spécifier le format du contenu attendu par le client.

`readEntity(String.class)` permet de récupérer le corps de la réponse dans un objet de type `String`.

- Il faut ensuite utiliser l'API JSON-P, pour parser le tableau json représentant la liste des paiements afin d'alimenter la liste Java `paymentOrders`

```
try(JsonReader jreader = Json.createReader(new StringReader(jsonContent))){
    //objet Java représentant un tableau json
    JSONArray jArray = jreader.readArray();
    for(int i = 0; i < jArray.size(); i++){ //pour chaque entrée du tableau
        JsonObject jObject = jArray.getJsonObject(i); //on récupère l'objet json
        //on récupère la valeur de chaque donnée
        Long id = jObject.getJsonNumber("id").longValue();
```

```

        Double amount = jsonObject.getJsonNumber("amount").doubleValue();
        //on construit l'URL localisant un ordre de paiement
        String location = baseUrl+"/"+id;
        //on alimente la liste avec un ordre de paiement
        paymentOrders.add(new PaymentOrder(location, amount, id));
    }
}

```

A partir d'un `JsonReader` chargé avec le contenu json de la réponse, on obtient un objet Java représentant un tableau json (`JSONArray`). On parcourt ce tableau pour extraire chaque entrée en tant que `JsonObject` (représentation Java d'un objet json). Chaque objet json va nous permettre de construire une instance `PaymentOrder` qu'on ajoute à la liste `paymentOrders`.

- Enfin créez une méthode `void init()` annotée avec `@PostConstruct` dans laquelle vous invoquez simplement `loadAllPayments` afin d'initialiser le bean avec la liste des paiements.

Voici un extrait de votre bean :

```

@Named(value = "paymentOrderModel")
@RequestScoped
public class PaymentOrderBean{

    private final String baseUrl = "http://localhost:11080/bankFacade-war/banking/payment";
    private List<PaymentOrder> paymentOrders = new ArrayList<>();

    @PostConstruct
    void init(){
        loadAllPayments();
    }
    ...

    private void loadAllPayments(){
        paymentOrders.clear();

        Client client = ClientBuilder.newClient();
        WebTarget target = client.target(baseUrl).path("payments");
        Response resp = target.request().accept(MediaType.APPLICATION_JSON_TYPE).get();
        String jsonContent = resp.readEntity(String.class);
        resp.close();
        client.close();

        try(JsonReader jreader = Json.createReader(new StringReader(jsonContent))){
            JSONArray jArray = jreader.readArray();
            for(int i = 0; i < jArray.size(); i++){

```

```

        JsonObject jObject = jsonArray.getJSONObject(i);
        Long id = jObject.getLong("id").longValue();
        Double amount = jObject.getDouble("amount").doubleValue();
        String location = baseUrl + "/" + id;
        paymentOrders.add(new PaymentOrder(location, amount, id));
    }
}
}

```

Note : Bien évidemment dans une vraie application vous ne chargeriez pas l'ensemble des ordres de paiements du fait que le volume aurait des conséquences sur les performances de l'application.

Pour éviter cela vous pourriez utiliser le principe HATEOAS afin d'embarquer dans vos réponses des « liens » permettant de naviguer dans le jeu d'enregistrements représentant les paiements. Autrement-dit vous pourriez paginer les résultats retournés pour éviter le chargement d'un seul bloc. HATEOAS est en gros un principe architectural http et donc REST spécifiant comment changer l'état de l'application grâce à des informations (ex : des liens) embarquées dans la réponse. Ici le changement d'état correspondrait à une navigation de type « page suivante/prédédente » dans un jeu de données. Pour plus d'informations (vous ne devriez pas avoir de mal à trouver des liens sur le sujet) :

<http://restcookbook.com/Basics/hateoas/>

<https://en.wikipedia.org/wiki/HATEOAS>

88. Créez dans un dossier admin sous Web Pages la page JSF (technologie Facelets)

paymentAdmin.xhtml.

Cette vue listera l'ensemble des ordres de paiement et permettra d'en supprimer.

Voici un extrait présentant le code à ajouter dans admin/paymentAdmin.xhtml :

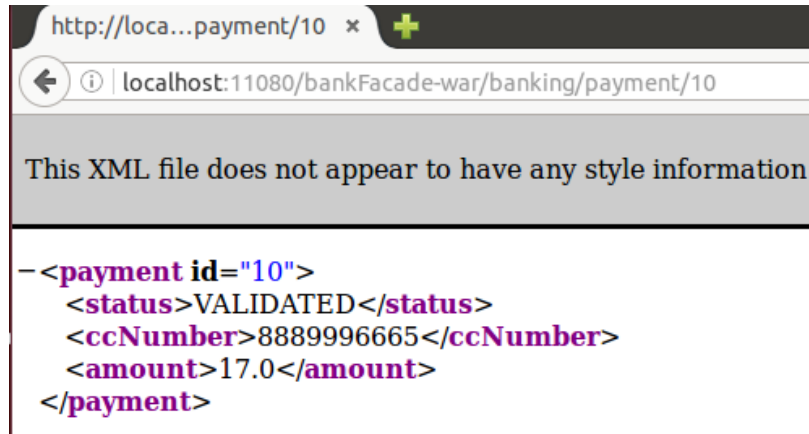
```

<h:body>
  <h:form>
    <h:dataTable value="#{paymentOrderModel.paymentOrders}" var="po">
      <h:column>
        <f:facet name="header">N° ordre paiement</f:facet>
        <h:outputText value="#{po.orderNumber}"/>
      </h:column>
      <h:column>
        <f:facet name="header">Montant paiement</f:facet>
        <h:outputText value="#{po.amount} €"/>
      </h:column>
      <h:column>
        <f:facet name="header">Adresse paiement</f:facet>
        <h:outputLink value="#{po.location}">#{po.location}</h:outputLink>
      </h:column>
    </h:dataTable>
  </h:form>

```

</h:body>

La balise `outputLink` représente un lien hypertexte classique. Elle vous permettra de requêter en GET un ordre de paiement donné. Il ne s'agit pas d'une requête JSF mais d'une requête externe du navigateur vers une ressource. La plupart des navigateurs (dont Mozilla Firefox) acceptent comme type de média par défaut « XML ». Le paiement requêté sera donc représenté au format XML dans le navigateur :



Côté application REST, c'est la méthode `PaymentResource.getStoredPayment` qui sera invoquée pour traiter cette demande GET.

89. Dans le fichier `WEB-INF/faces-config.xml` de `webStore` ajoutez la règle de navigation pour accéder à `admin/paymentAdmin.xhtml` depuis la page `payment.xhtml`. La sortie permettant la navigation vers la « console d'administration » est la chaîne **admin**.

Extrait du fichier `faces-config.xml` avec la nouvelle règle de navigation en gras :

```
<navigation-rule>
  <from-view-id>/payment.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>valid</from-outcome>
    <to-view-id>/success.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>nonvalid</from-outcome>
    <to-view-id>/error.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>admin</from-outcome>
    <to-view-id>/admin/paymentAdmin.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

90. Enfin dans la page `payment.xhtml` ajoutez une balise JSF `commandLink` pour naviguer vers `admin/paymentAdmin.xhtml` :

```
...
<h:commandButton value="Payer avec Rest"
action="#{paymentModel.doPaymentWithRest}"/>
    <br/>
    <h:commandLink action="admin" value="gestion des paiements"/>
</h:form>
...
```

91. Lancez un Clean & Build puis un Run sur le projet `webStore` pour tester l'affichage des ordres de paiement. En cliquant sur l'hyperlien d'un ordre de paiement vous accéderez à la structure de celui-ci (comme indiqué précédemment).

92. Dans **`PaymentOrderBean`**, implémentez la méthode `public String cancelPayment(Long id)`. Pour rappel, depuis JSF 2 (Java EE 6), les méthodes d'action acceptent des paramètres. Le paramètre `id` correspond à l'ordre de paiement à supprimer.

Pour supprimer un ordre de paiement, il faut exécuter une requête `http DELETE` à l'adresse de du paiement. Le pattern d'URI à considérer est :

<http://localhost:11080/bankFacade-war/banking/payment/{id}>

Voici le code de la méthode :

```
public String cancelPayment(Long id){
    Client client = ClientBuilder.newClient();
    WebTarget target = client.target(baseUrl).path("{id}").resolveTemplate("id", id);
    target.request().delete();
    client.close();
    loadAllPayments();
    return null;
}
```

`resolveTemplate()` permet d'assigner au template parameter spécifié via **`path()`** une valeur en vue d'exécuter la requête de déletion.

Notez qu'on n'utilise pas ici par simplicité la réponse retournée par l'exécution de la requête `DELETE`.

La méthode `cancelPayment` retournant `null`, l'action de déletion n'entraînera pas de navigation vers une nouvelle vue.

93. Ajoutez dans la page `admin/paymentAdmin.xhtml`, le bouton (`commandButton` JSF) permettant d'exécuter une requête de déletion d'un paiement localisé à une adresse donnée :

```
...
<h:column>
```

```
<f:facet name="header">Adresse paiement</f:facet>
  <h:outputLink value="#{po.location}">#{po.location}</h:outputLink>
</h:column>
<h:column>
  <h:commandButton value="supprimer"
action="#{paymentOrderModel.cancelPayment(po.orderNumber)}"/>
</h:column>
</h:dataTable>
....
```

Notez que l'argument passé à *cancelPayment* est la valeur retournée par *po.getOrderNumber()* qui correspond à l'id d'un paiement.

94. Lancez un Clean & Build puis un Run sur le projet webStore pour tester l'ensemble depuis la création SOAP et REST de paiements jusqu'à la déléition de paiements.

Voici le type de vue « admin » que vous devriez avoir :

N° ordre paiement	Montant paiement	Adresse paiement	
2	130.0 €	http://localhost:11080/bankFacade-war/banking/payment/2	<input type="button" value="supprimer"/>
10	17.0 €	http://localhost:11080/bankFacade-war/banking/payment/10	<input type="button" value="supprimer"/>
13	13.33 €	http://localhost:11080/bankFacade-war/banking/payment/13	<input type="button" value="supprimer"/>
14	78.0 €	http://localhost:11080/bankFacade-war/banking/payment/14	<input type="button" value="supprimer"/>

Partie 3 : Mise en place de la communication asynchrone avec JMS

Création du domaine bank

95. Créez le domaine *bank* qui hébergera la logique du (pseudo) traitement du paiement bancaire.

La base pour les ports est 12000.

Utilisez par exemple le mot de passe **abank**.

Pour le mot de passe maître, laissez le mot de passe par défaut changeit. La procédure est la même que pour la création des 2 domaines précédents.

```
asadmin --user admin create-domain --portbase 12000 --savemasterpassword=true bank
```

96. Référez le domaine bank dans le nœud Servers de NetBeans. La manipulation est identique au référencement des 2 premiers domaines.

Domaine bank :

Nom d'instance : bank

Sélection du dossier d'installation de Payara comme précédemment

Domaine sélectionné : bank

Nom utilisateur : admin / mot de passe abank dans notre cas.



Création du MDB -- traitement asynchrone du paiement

Le web service « délègue » au Message-Driven Bean (MDB), plus précisément à l'infrastructure JMS, le traitement du paiement (logique complexe).

Domaine middleware

Ce domaine va héberger la destination de type queue pour les ordres de paiement.

97. Vous allez utiliser la fabrique de connexion JMS par défaut que tout produit Java EE 7 doit implémenter. Pour visualiser cette fabrique, ouvrez la console d'administration du domaine middleware (<http://localhost:11048>) puis naviguez dans *nœud JMS Resources* > *nœud Connection Factories*

Connection Factories (1)				
<input checked="" type="checkbox"/>	<input type="button" value="New..."/>	<input type="button" value="Delete"/>	<input type="button" value="Enable"/>	<input type="button" value="Disable"/>
Select	JNDI Name	Logical JNDI Name	Enabled	Resource Type
<input type="checkbox"/>	jms/_defaultConnectionFactory	java:comp/DefaultJMSConnectionFactory	<input checked="" type="checkbox"/>	javax.jms.ConnectionFactory

98. Créez la queue pour les ordres (messages) de paiement :

Si la description de votre de votre queue contient des caractères « spéciaux » tels que les accents, vous ne pourrez-plus (re)démarrer le domaine Payara depuis NetBeans. Pour démarrer dans ce cas, il faut utiliser la commande *asadmin start-domain nom-du-domaine*. C'est un problème du plugin Serveur NetBeans.
 Moralité : pas d'accents ni autres caractères spéciaux dans les descriptions.

Nœud *JMS Resources JMS>Destination Resources >New...*

JNDI Name : `jms/paymentQueue`

Physical Destination Name : `physPaymentQueue`

Resource Type : `javax.jms.Queue`

Cliquez sur OK pour finaliser.

New JMS Destination Resource

The creation of a new Java Message Service (JMS) destination resource also creates an admin object resource.

JNDI Name: *

Physical Destination Name *
Destination name in the Message Queue broker. If the destination does not exist, it will be created automatically when needed.

Resource Type: *

Description:

Status: ☒ Enabled

Si vous naviguez dans nœud *server (Admin Server) > onglet JMS Physical Destinations*, vous ne verrez pas la destination physique `phyPaymentQueue`. Elle sera créée à la demande par le fournisseur JMS intégré au serveur lors du premier envoi de message.

Dans la classe du Session Bean **BankingServiceBean**, vous allez implémenter l'envoi de messages JMS à la queue. Ces messages seront consommés par des instances d'un Message-Driven Bean (MDB) déployé dans le domaine *bank*. Vous allez utiliser l'API JMS 2.0 (paquetage `javax.jms`) simplifié pour envoyer l'ordre de Paiement à la queue.

99. Tout d'abord, Injectez le contexte JMS et injectez la queue préalablement créée :

```
@Stateless
@WebService(
    endpointInterface = "com.bank.paymentmgmt.facade.BankingServiceEndpointInterface",
    portName = "BankingPort",
    serviceName = "BankingService"
```

```

)
public class BankingServiceBean implements BankingServiceEndpointInterface,
BankingServiceRemote {

    @Inject
    private PaymentDAO paymentDAO;

    @Inject //paquetage javax.inject
    private JMSContext context; //paquetage javax.jms

    @Resource(lookup = "jms/paymentQueue") //paquetage javax.annotation
    private Queue paymentQueue; //paquetage javax.jms

    ....

```

Le contexte JMS injecté est géré par le container tout comme la queue. Le contexte JMS combine de nombreuses fonctionnalités de l'API classique JMS pour simplifier le codage.

100. Toujours dans BankingServiceBean, implémentez la méthode privée `void sendPayment(Payment payment)` encapsulant l'envoi d'un message JMS contenant les informations de paiement.
- Le type Payment est annoté avec des annotations JAX-B, il sera donc facilement convertible en message XML. Implémentez la conversion en XML du paiement grâce à l'API de flux JAX-B :

```

private void sendPayment(Payment payment){
    //utilisation de l'API JAX-B de gestion de flux pour marshaller (transformer) l'objet
    //Payment en chaine XML
    JAXBContext jaxbContext;
    try {
        //obtention d'une instance JAXBContext associée au type Payment annoté avec JAX-B
        jaxbContext = JAXBContext.newInstance(Payment.class);
        //création d'un Marshaller pour transformer l'objet Java en flux XML
        Marshaller jaxbMarshaller = jaxbContext.createMarshaller();

        StringWriter writer = new StringWriter();

        //transformation de l'objet en flux XML stocké dans un Writer
        jaxbMarshaller.marshal(payment, writer);
        String xmlMessage = writer.toString();
        //affichage du XML dans la console de sortie
        System.out.println(xmlMessage);

    } catch (JAXBException ex) {

```

```

        Logger.getLogger(BankingServiceBean.class.getName()).log(Level.SEVERE, null, ex);
    }
}

```

- Complétez la méthode **sendPayment** pour envoyer l'ordre de paiement à la queue. Il faut créer via le contexte JMS un message de type **TextMessage** qui encapsule xmlMessage puis il faut envoyer le message à la queue. Ci-dessous un extrait de la méthode avec les instructions d'envoi de message JMS en gras :

```

private void sendPayment(Payment payment){
    //utilisation de l'API JAX-B de gestion de flux pour marshaller (transformer) l'objet
    //Payment en chaine XML
    ...
    //affichage du XML dans la console de sortie
    System.out.println(xmlMessage);
    //encapsulation du paiement au format XML dans un objet javax.jms.TextMessage
    TextMessage msg = context.createTextMessage(xmlMessage);

    //envoi du message dans la queue paymentQueue
    context.createProducer().send(paymentQueue, msg);

    } catch (JAXBException ex) {
        Logger.getLogger(BankingServiceBean.class.getName()).log(Level.SEVERE, null, ex);
    }
}

```

- Tout ordre de paiement créé ou supprimé doit être envoyé à la destination JMS en vue d'être traité.
 - o Complétez la méthode *BankingServiceBean.createPayment* pour envoyer sous forme de message JMS l'ordre de paiement créé :

```

@Override
public Boolean createPayment(String ccNumber, Double amount) {
    if(ccNumber.length()== 10 ){
        System.out.println("Montant payé : "+amount + " €");
        Payment p = new Payment();
        p.setCcNumber(ccNumber);
        p.setAmount(amount);
        p = paymentDAO.add(p);

        sendPayment(p); //envoi du paiement sous forme de message JMS
                        //formaté en XML

        return true;
    } else {
        return false;
    }
}

```

```
}
}
```

- Modifiez la méthode *BankingServiceBean.deleteStoredPayment* pour envoyer les informations de déletion :

```
@Override
public Payment deleteStoredPayment(Long id) {
    Payment p = paymentDAO.delete(id);
    if(p!=null){
        sendPayment(p);
    }
    return p;
}
```

Vous avez donc utilisé l'API simplifié de JMS 2.0 permettant d'envoyer un message JMS. Votre session bean fait donc office de producteur de message.

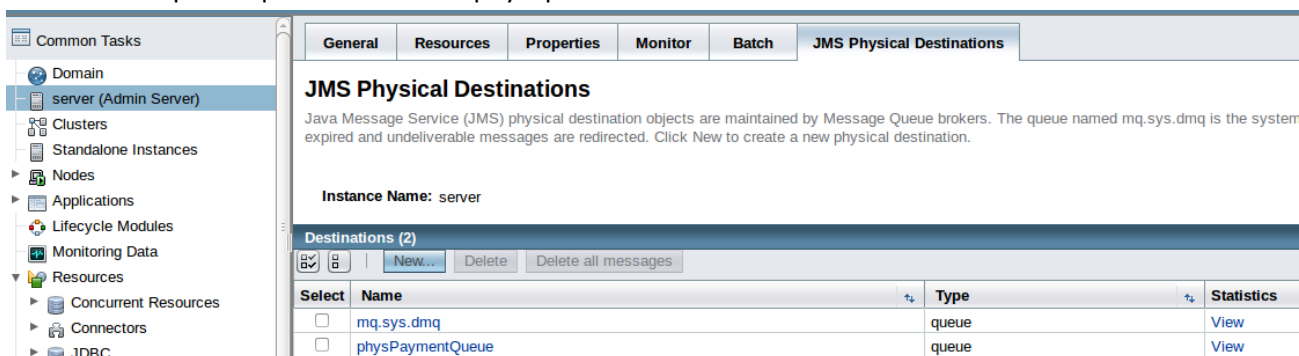
101. Redéployez (*Run*) le module EJB **bankFacade-ejb**, si ce n'est fait automatiquement par NetBeans après sauvegarde, et testez depuis webStore la création et la suppression d'ordres de paiement. Pour que le test envoie dans la queue, il faut saisir 10 chiffres.

Notez que lors du redéploiement une nouvelle instance de MapPaymentDAO (de scope Application) est créée avec une Map vide.

Vérifiez l'état de la queue grâce à la console d'administration du domaine middleware :

Server (Admin Server)> JMS Physical Destinations pour accéder aux queues physiques créées par le provider JMS.

Vous remarquerez que la destination physique a été créée.



JMS Physical Destinations

Java Message Service (JMS) physical destination objects are maintained by Message Queue brokers. The queue named mq.sys.dmq is the system expired and undeliverable messages are redirected. Click New to create a new physical destination.

Instance Name: server

Select	Name	Type	Statistics
<input type="checkbox"/>	mq.sys.dmq	queue	View
<input type="checkbox"/>	physPaymentQueue	queue	View

En cliquant sur *View* et en descendant dans les statistiques, vous pourrez vérifier que des messages sont bien en attente de consommation dans la queue :

Average Number of Backup Consumers	0	Average number of associated backup message consumers since broker started
Number of Messages	4	Current number of messages stored in memory and persistent store. This value does
Number of Remote Messages	0	Current number of messages stored in memory and persistent store that were produ
Number of Messages Pending Acknowledgment	0	

Ici on a créé 3 ordres de paiements et supprimé 1 ordre de paiement, il y a donc 4 messages JMS qui ont été envoyés dans la queue JMS.

102. Dans la console Web du domaine **middleware** naviguez dans *Configurations > server-config > System Properties* et notez le port d'écoute JMS (JMS_PROVIDER_PORT). Ce devrait être **11076**.

Additional Properties (1)		
<input checked="" type="checkbox"/>	<input type="text" value="JMS_PROVIDER_PORT"/>	11076

Fermez la console web du domaine middleware. Le domaine **middleware** ne doit pas être arrêté.

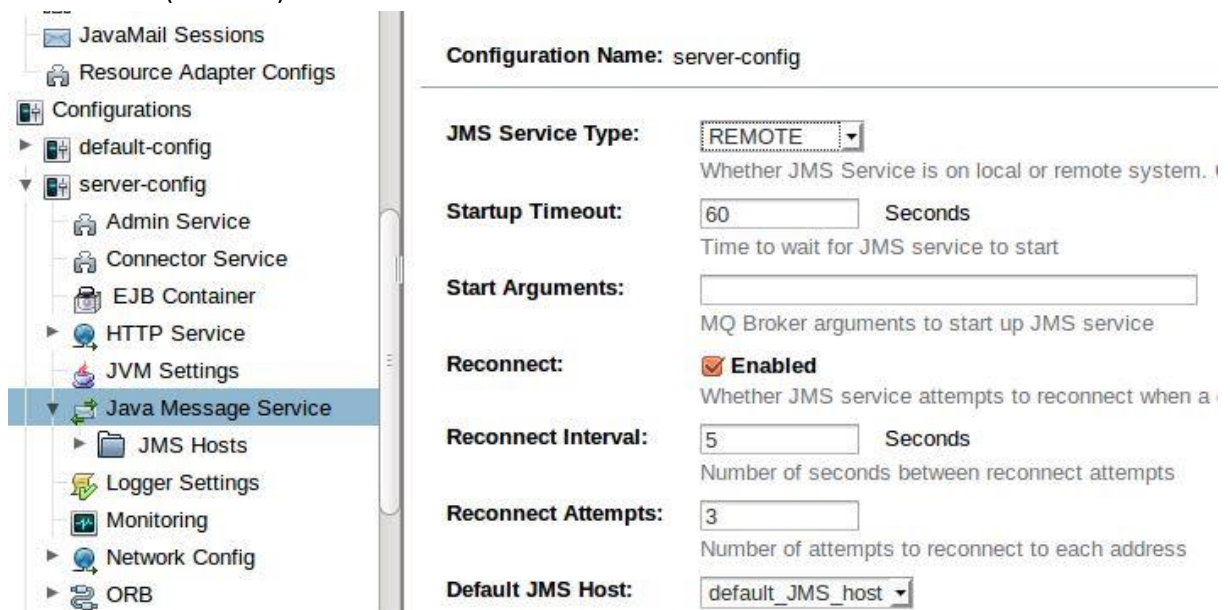
Domaine bank :

L'objectif est la création du Message-Driven Bean consommant l'ordre de paiement crée ou supprimé envoyé dans la queue jms/paymentQueue.

Vous désirez utiliser un provider JMS se situant dans une instance de domaine différente. Dans votre cas, il s'agit du service configuré dans le domaine middleware

Passage un peu critique donc suivez attentivement les étapes

102. Démarrez, si nécessaire, le domaine bank. Dans la console web du domaine bank, au niveau du nœud *Configurations>server-config> Java Message Service*. Précisez que le service JMS est distant (REMOTE).



The screenshot shows the JBoss console with the following configuration for the Java Message Service:

- Configuration Name:** server-config
- JMS Service Type:** REMOTE (Whether JMS Service is on local or remote system.)
- Startup Timeout:** 60 Seconds (Time to wait for JMS service to start)
- Start Arguments:** (MQ Broker arguments to start up JMS service)
- Reconnect:** Enabled (Whether JMS service attempts to reconnect when a)
- Reconnect Interval:** 5 Seconds (Number of seconds between reconnect attempts)
- Reconnect Attempts:** 3 (Number of attempts to reconnect to each address)
- Default JMS Host:** default_JMS_host


Enregistrez la modification (bouton *save* en haut à droite). Surtout ne redémarrez pas encore le domaine **bank**. Si vous redémarrez, le provider JMS distant ne sera pas trouver.

103. Toujours dans la console du domaine **bank**, développez le nœud *Configurations>server-config>*

System Properties

Renseignez la valeur par défaut du port d'écoute du provider JMS. Il s'agit du port du provider JMS s'exécutant dans le domaine **middleware**. C'est le port noté à l'étape **102**. Dans la copie d'écran ci-dessous 18676 est spécifié car le provider JMS auquel nous voulons accéder s'exécute dans middleware et écoute sur **11076**.


Enregistrez la modification (bouton *save*).

Additional Properties (1)		
		
<input type="checkbox"/>	<input type="text" value="JMS_PROVIDER_PORT"/>	11076

Attention :

NE PAS EFFECTUER LA MISE A JOUR DU MOT DE PASSE DEMANDEE PAR LE NAVIGATEUR (s'il y a demande)

104. Le port changé, retournez dans le nœud Java Message Service sous server-config et cliquez sur le bouton **ping**. Le message Ping succeeded devrait s'afficher.

 Ping Succeeded

Java Message Service

General properties for the Java Message Service (JMS) service apply only to the application server's default JMS provider, GlassFish Message Queue. All other properties apply to the Connector Resources screens.

 Ping

Sous *server (Admin Server) > JMS Physical Destinations* de ce domaine **bank**, vous verrez référencée la queue physique distante *physPaymentQueue* du domaine **middleware**.

105. Fermez la console web et redémarrez depuis NetBeans l'instance **bank**. Après redémarrage du domaine **bank**, ouvrez de nouveau la console web de bank pour vérifier que le service JMS de **bank** pointe toujours bien vers le service distant JMS de **middleware**.

106. Dans le domaine **bank**, créez la queue pour les messages de paiement comme vous l'avez fait précédemment pour le domaine **middleware** (*JMS Resources > JMS Destinations > New ...*) :

- JNDI Name : *jms/paymentQueue*
- Pointe vers la destination physique (Physical Destination Name) : *physPaymentQueue*
- Resource Type : *javax.jms.Queue*

Cet objet administré de Payara, qui est enregistré sous le nom *jms/paymentQueue*, pointe vers la queue physique *physPaymentQueue* de **middleware**.

Cliquez sur OK pour finaliser la création de la destination.

JNDI Name: *	<input type="text" value="jms/paymentQueue"/>
Physical Destination Name *	<input type="text" value="physPaymentQueue"/> <small>Destination name in the Message Queue broker. If the destination does not exist, it</small>
Resource Type: *	<input type="text" value="javax.jms.Queue"/>
Description:	<input type="text" value="destination logique pointant vers le broker de middleware"/>
Status:	<input checked="" type="checkbox"/> Enabled

107. Créez un nouveau module EJB Maven d'entreprise nommé **bankBusiness-ejb** dans le dossier contenant vos autres projets. Liez ce dernier au domaine **bank**.

- Spécifiez le group Id : com.bank
- Spécifiez le paquetage : com.bank.paymentmgmt.logic
- Associez-le au domaine **bank**.

Steps	Name and Location
1. Choose Project	Project Name: <input type="text" value="bankBusiness-ejb"/>
2. Name and Location	Project Location: <input type="text" value="/home/cesi/javaProjects/v2017/v1"/> <input type="button" value="Browse..."/>
3. Settings	Project Folder: <input type="text" value="cesi/javaProjects/v2017/v1/bankBusiness-ejb"/>
	Artifact Id: <input type="text" value="bankBusiness-ejb"/>
	Group Id: <input type="text" value="com.bank"/>
	Version: <input type="text" value="1.0-SNAPSHOT"/>
	Package: <input type="text" value="com.bank.paymentmgmt.logic"/> (Optional)

Supprimez le session bean NewSessionBean créé par Maven lors de la création du projet.

108. Modifiez le fichier pom.xml situé sous Project Files :

- Donnez un nom final à l'archive : bankBusiness-ejb
- Changez la version Java utilisée par Maven. Java 8 est utilisé.
- Précisez au niveau du plugin EJB que c'est la version 3.2 (Java EE 7) qui est utilisée.

Voici un extrait du pom.xml

```
...
<build>
  <finalName>bankBusiness-ejb</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
```



```
<artifactId>maven-compiler-plugin</artifactId>
<version>3.1</version>
<configuration>
  <source>1.8</source>
  <target>1.8</target>
  ...
</configuration>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-ejb-plugin</artifactId>
  <version>2.3</version>
  <configuration>
    <ejbVersion>3.2</ejbVersion>
  </configuration>
</plugin>
...
```

109. Créez dans le paquetage *com.bank.paymentmgmt.logic* le Message-Driven Bean (le consommateur de message) nommé *PaymentProcessor* :

Clic <droite> sur le module> New...> Sélectionnez Message-Driven Bean.

EJB Name : PaymentProcessor.

Cochez *Server Destinations* et sélectionnez *jms/paymentQueue*. Et cliquez sur Next >

L'écran suivant permet de sélectionner les propriétés de configuration du MDB. Ne modifiez pas les propriétés :

New File

Steps

1. Choose File Type
2. Name and Location
- 3. Activation Config Properties**

Activation Config Properties

Property Name	Property Value
acknowledgeMode	AUTO_ACKNOWLEDGE
clientId	
connectionFactoryLookup	
destinationType	QUEUE
destinationLookup	.jms/paymentQueue
messageSelector	
subscriptionDurability	NON_DURABLE
subscriptionName	

Enfin cliquez sur Finish.

Votre squelette classe MDB devrait être le suivant :

```
package com.bank.paymentmgmt.logic;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue =
    "jms/paymentQueue"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
    "javax.jms.Queue")
})
public class PaymentProcessor implements MessageListener {

    public PaymentProcessor() {
    }

    @Override
    public void onMessage(Message message) {
    }

}
```

Le container surveille l'arrivée de message dans la destination de type queue. Lorsqu'un message est disponible il assigne une instance du pool du MDB PaymentProcessor et invoque la méthode « écouteur » onMessage chargée de traiter le message.

Vous remarquerez que les annotations de configuration de votre MDB ne contiennent pas la propriété `acknowledgeMode`. Comme vous utilisez le système par défaut de transactions gérées par le container, l'acquittement du message auprès du fournisseur JMS sera effectué par le container dès que la méthode aura fini de s'exécuter. Si la transaction échoue, le container redélivra le message. Cette propriété est utile quand la démarcation des transactions n'est pas gérée par le container. Dans ce cas, le message n'étant pas enrôlé dans une transaction active, il faut préciser le mode d'acquittement. La propriété `acknowledgeMode` étant ici inutile, NetBeans ne l'a donc pas incorporée à la configuration. Dans le même ordre d'idée, la propriété `subscriptionDurability` n'est pas spécifiée car elle ne concerne que le modèle Publish-and-Subscribe de JMS. Or vous travaillez selon un modèle Point-to-Point.

110. Implémentez la méthode **`onMessage(Message message)`** de façon à ce que l'on puisse vérifier dans la console de sortie de **bank** (output dans NetBeans) que le message a bien été reçu. Voici le code pour consommer l'ordre de paiement :

```
@Override
public void onMessage(Message message) {
    try {
        //on extrait le paiement du corps du message. - getBody est une méthode JMS 2.0
        String paymentMessage = message.getBody(String.class);
        for(int i = 0; i<=40;i++){
            System.out.println("[traitement long d'integration dans le processus bancaire de]");
            System.out.println(paymentMessage);
        }
        System.out.println("l'ordre de paiement "+paymentMessage+" va être retiré de la queue");
    } catch (JMSEException ex) {
        Logger.getLogger(PaymentProcessor.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Les boucles `for` ne sont là que pour donner un effet « traitement long ». Ici on se limite à l'affichage du message de type texte représentant un paiement au format XML.

`JMSEException` appartient bien évidemment au paquetage `javax.jms` et `Logger` et `Level` appartiennent au paquetage `java.util.logging`.

111. Exécutez en cliquant sur *Run* votre projet `bankBusiness-ejb`. Normalement vous devriez voir apparaître dans la console de sortie de **bank** le traitement des paiements créés et supprimés.

```
Info: bankBusiness-ejb was successfully deployed in 2,699 milliseconds.
Info: [traitement long d'integration dans le processus bancaire de]
Info: <?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="4"><status>VALIDATED</status><ccNumber>7894563571</ccNumber><amount>29.99</amount></payment>
Info: [traitement long d'integration dans le processus bancaire de]
Info: <?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="4"><status>VALIDATED</status><ccNumber>7894563571</ccNumber><amount>29.99</amount></payment>
Info: [traitement long d'integration dans le processus bancaire de]
Info: <?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="4"><status>VALIDATED</status><ccNumber>7894563571</ccNumber><amount>29.99</amount></payment>
Info: [traitement long d'integration dans le processus bancaire de]
Info: <?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="4"><status>VALIDATED</status><ccNumber>7894563571</ccNumber><amount>29.99</amount></payment>
Info: [traitement long d'integration dans le processus bancaire de]
Info: <?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="4"><status>VALIDATED</status><ccNumber>7894563571</ccNumber><amount>29.99</amount></payment>
Info: [traitement long d'integration dans le processus bancaire de]
Info: <?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="4"><status>VALIDATED</status><ccNumber>7894563571</ccNumber><amount>29.99</amount></payment>
Info: <?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="4"><status>VALIDATED</status><ccNumber>7894563571</ccNumber><amount>29.99</amount></payment>
Info: <?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="4"><status>VALIDATED</status><ccNumber>7894563571</ccNumber><amount>29.99</amount></payment>
```

S'il y a plusieurs messages en queue, le container EJB mettra à disposition simultanément plusieurs instances de PaymentProcessor pour que chacune d'elles traite un message (Une instance ne traite qu'un seul message à la fois). Aussi dans la console de sortie de bank vous verrez peut-être des logs mélangés car plusieurs instances consomment simultanément des messages de la queue.

Le paiement une fois intégré au processus bancaire, le modèle Request / Reply JMS pourrait être utilisé pour envoyer un message de réponse à la façade pour l'informer que le paiement est pris en charge par le système bancaire et que celui-ci peut être supprimer de la "base"(simulée par une simple Map dans notre prototype) ou archiver.

Une transaction bancaire est un processus complexe nécessitant la communication entre de nombreuses applications. Le MDB n'est qu'un service intervenant en début de processus.

Parmi les tâches de ce processus on retrouvera la détection des fraudes, les demandes d'autorisations auprès des banques, le traitement par lots du transfert des paiements...

112. Testez de nouveau votre application distribuée dans son intégralité pour bien mettre en avant le traitement asynchrone JMS et la fiabilité de traitement du message (aucun ordre de paiement n'est perdu) :

Stoppez l'instance **bank**.

Saisissez un nouvel ordre de paiement (avec un numéro de carte valide) depuis le formulaire de paiement de *webStore*. Testez aussi la suppression d'ordre de paiements

Puis redémarrez l'instance **bank**.

Les paiements seront bien consommés par le MDB.

Partie 4 : Distribution physique de l'application

Vous allez maintenant distribuer physiquement l'application. Pour cela vous devez déployer depuis un poste de déploiement chacun des 3 projets sur une machine différente. Il y a 4 machines au total.

Vous allez attribuer des IP fixe à chacun des postes. Si vous êtes en VM vous pouvez ajouter une seconde carte réseau de type pont (bridge) ou réseau privé hôte (host-only network) uniquement si l'ensemble des Vms est localisé sur la même machine hôte. Avec le mode bridge, vous pourrez faire communiquer des Vms localisés sur des postes hôtes différents.

Conseil : Faites une sauvegarde de vos projets au cas où. Surtout webStore qui sera en fait le seul projet dont le code sera modifié. Vous pouvez aussi sauvegarder vos domaines locaux : Pour cela rendez-vous dans l'annexe en fin de document.

113. Veuillez-vous assurer de l'infrastructure physique suivante (ou approchante) :

- a) Poste qui héberge l'application web invoquant les web services de paiement.
Le domaine Payara concerné est **store**.
Nous considérons pour la suite que le poste s'appelle **storeSRV** et qu'il a l'adresse IP **192.168.2.10/255.255.255.0**.
- b) Machine qui héberge le web service et le provider JMS.
Le domaine Payara concerné est **middleware**.
Nous considérons pour la suite que le poste s'appelle **frontSRV** et qu'il a l'adresse IP **192.168.2.11/255.255.255.0**.
- c) Poste destiné à héberger le MDB qui consommera la queue située sur le poste distant frontSRV.
Le domaine Payara concerné est **bank**.
Nous considérons pour la suite que le poste s'appelle **backSRV** et qu'il a l'adresse IP **192.168.2.12/255.255.255.0**.
- d) Poste de déploiement : poste sur lequel NetBeans est installé et donc depuis lequel vous allez déployer les projets.
Nous considérons pour la suite que le poste s'appelle **prototype** et qu'il a l'adresse IP **192.168.2.14/255.255.255.0**

Notez que dans Ubuntu, le nom de la machine est défini dans le fichier /etc/hostname et est aussi référencé dans /etc/hosts. Si vous changez de nom de machine, il faut mettre à jour ces 2 fichiers avec le nouveau nom.

114. Connectez les 4 postes sur un switch si vous travaillez bien évidemment sur plusieurs ordinateurs.

115. Modifiez le fichier hosts de **frontSRV**.

Pour linux : /etc/hosts

Pour Windows : C:\Windows\System32\drivers\etc\hosts

Si dessous un exemple de fichier hosts Linux. Ce qui est à ajouter est présenté en gras.

Sans la déclaration de frontSRV et backSRV, les queues physiques hébergées par middleware

Ne sont affichées pas dans la console web.

```
127.0.0.1      localhost
#127.0.1.1    frontSRV // à commenter
192.168.2.11  frontSRV
192.168.2.12  backSRV
# The following lines are desirable for IPv6 capable hosts
::1          ip6-localhost ip6-loopback
fe00::0      ip6-localnet
ff00::0      ip6-mcastprefix
ff02::1      ip6-allnodes
ff02::2      ip6-allrouters
```

116. Modifiez le fichier hosts de **backSRV**. Sans la déclaration de frontSRV, ce nom ne pourra pas être utilisé dans la configuration JMS du domaine **bank** pour contacter le provider JMS distant hébergé sur *frontSRV*. Même remarque que précédemment pour l'affichage des queues physiques dans la console web de bank.

```
127.0.0.1      localhost
#127.0.1.1    backSRV //à commenter
192.168.2.11  frontSRV
192.168.2.12  backSRV

# The following lines are desirable for IPv6 capable hosts
::1          ip6-localhost ip6-loopback
fe00::0      ip6-localnet
ff00::0      ip6-mcastprefix
ff02::1      ip6-allnodes
ff02::2      ip6-allrouters
```

117. Modifiez le fichier hosts de **storeSRV** :

```
127.0.0.1      localhost
#127.0.1.1    storeSRV // à commenter
192.168.2.10  storeSRV
192.168.2.11  frontSRV
...
```

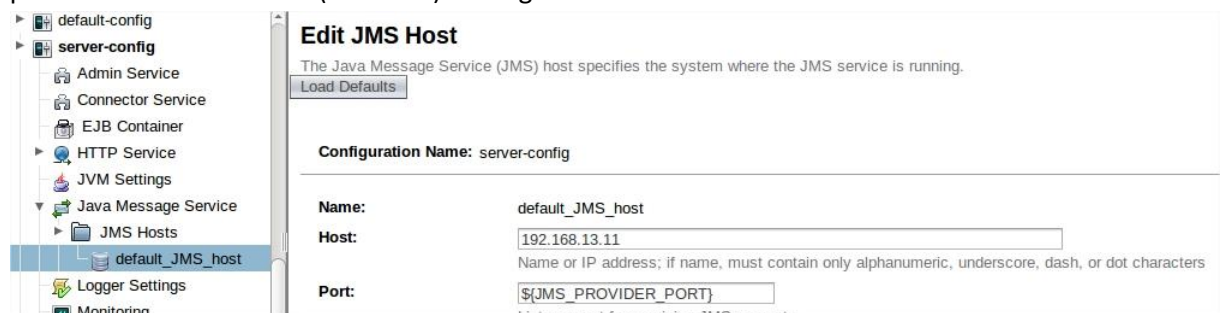
118. Modifiez le fichier hosts de prototype :

```
127.0.0.1      localhost
#127.0.1.1    prototype //à commenter
192.168.2.14   prototype
192.168.2.10   storeSRV
192.168.2.11   frontSRV
192.168.2.12   backSRV
...
```

119. Démarrez les domaines **bank** et **middleware** respectivement sur backSRV et frontSRV.

120. Faites pointer le domaine **bank** (bankSRV) vers le provider JMS de middleware (frontSRV).

Pour cela, ouvrez la console web du domaine **bank** et dans *server-config* > *Java Message Service* > *JMS Hosts* > *Default_JSM_Host*, remplacez localhost par l'IP de frontSRV (192.168.2.11) ou par le nom de la machine (frontSRV) hébergeant le service JMS :



OU

Edit JMS Host

The Java Message Service (JMS) host specifies the system where the JMS service is running.

[Load Defaults](#)

Configuration Name: server-config

Name: default_JMS_host

Host:

Name or IP address; if name, must contain only alphanumeric, underscore, dash, or dot characters

Profitez-en pour vous assurer que le service JMS du domaine **bank** est bien positionné sur REMOTE et que le port d'écoute est bien celui de middleware (dans ce tutoriel c'est 18676).

Une fois la configuration finalisée effectuez un ping depuis la page d'administration Web *Java Message Service* pour vérifier la communication.

121. Arrêtez puis redémarrez l'instance du domaine bank avec asadmin :

```
asadmin stop-domain bank
```

Puis

```
asadmin start-domain bank
```

*Si vous rencontrez des problèmes, supprimez `physPaymentQueue` au niveau du domaine **bank** et recommencez cette étape de redémarrage. Vous pouvez utiliser la commande `asadmin restart-domain` au lieu d'arrêter puis redémarrer le domaine*

122. Assurez-vous que, sur la machine (prototype) depuis laquelle vous allez déployez les projets, vos domaines « locaux » sont éteints.

Note : Si les machines hébergeant les domaines sont sous un environnement Linux, vous pouvez switcher en mode console pour les étapes suivantes. Cela vous fera économiser de la ressource mémoire, surtout si la maquette est déployée sous forme de VMs sur un même ordinateur. De toute façon vous pouvez accéder aux consoles web d'administration des domaines depuis le poste prototype.

Pour Ubuntu :

Ctrl + Alt + F6 pour switcher du mode graphique au mode console.

Ctrl + Alt + F7 pour switcher du mode console au mode graphique.

123. Démarrez, si ce n'est déjà fait, sur chacune des machines storeSRV, frontSRV et backSRV les domaines hébergés (respectivement store, middleware et bank).

124. Activez la capacité remote admin (via l'activation de l'accès sécurisé) pour chaque domaine distant que vous désirez joindre. Les domaines doivent être démarrés.

Sur la machine **hébergeant** le domaine que vous voulez rendre accessible à distance, vous aller utiliser la commande :

```
asadmin enable-secure-admin --port [NUMERO_PORT]
```

[NUMERO_PORT] correspond au numéro de port administratif du domaine que vous désirez rendre accessible à distance.

- Sur la machine storeSRV hébergeant le webStore, activez la capacité remote pour le domaine **store** en exécutant depuis la machine de déploiement :

```
asadmin enable-secure-admin --port 10048
```

Entrez le compte : admin.

Entrez le mot de passe : astore (par ex.).

Redémarrez le domaine store.

- Sur la machine frontSRV hébergeant le service Web, activez la capacité remote pour le domaine **middleware** en exécutant :

```
asadmin enable-secure-admin --port 11048
```

Entrez le compte : admin.

Entrez le mot de passe : amiddle (par ex.).

Redémarrez le domaine middleware.

- Sur la machine backSRV hébergeant le Message-Driven Bean, activez la capacité remote pour le domaine **bank** en exécutant :

```
asadmin enable-secure-admin --port 12048
```

Entrez le compte : admin.

Entrez le mot de passe : abank (par ex.).

Redémarrez le domaine bank.

Note : pour activer la capacité « remote admin », il faut obligatoirement un mot de passe attribué au compte admin.

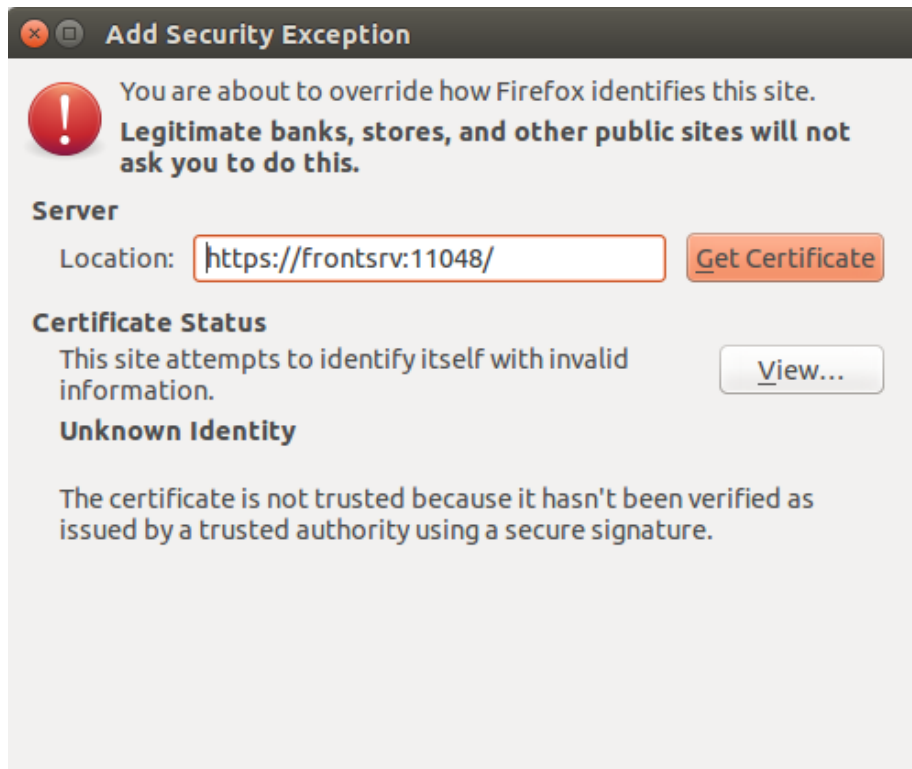
Désormais vous pouvez accéder aux consoles web d'administration des 3 domaines depuis la machine prototype.

- Pour accéder à la console du domaine store localisé sur storeSRV :
<https://192.168.2.10:10048/> ou <https://storeSRV:10048/>
- Pour accéder à la console du domaine middleware localisé sur frontSRV :
<https://192.168.2.11:11048/> ou <https://frontSRV:10048/>
- Pour accéder à la console du domaine bank localisé sur backSRV :
<https://192.168.2.12:12048/> ou <https://frontSRV:10048/>

Vous pouvez utiliser les noms de machines au lieu des IP si le fichier hosts de la machine depuis laquelle vous accédez à la console web distante est configuré avec les bonnes résolutions de noms.

Remarque : La sécurité ayant été activée, Il faudra ajouter dans le navigateur web l'autorisation d'accès https à la console Web d'administration.

Par exemple pour Mozilla, cliquez sur *I Understand the Risks*, puis sur le bouton *Add Exceptions...*, dans la fenêtre des exceptions de sécurité, cliquez sur *Confirm Security Exception*.



125. Dans la machine prototype, référez les 3 domaines distants dans NetBeans.

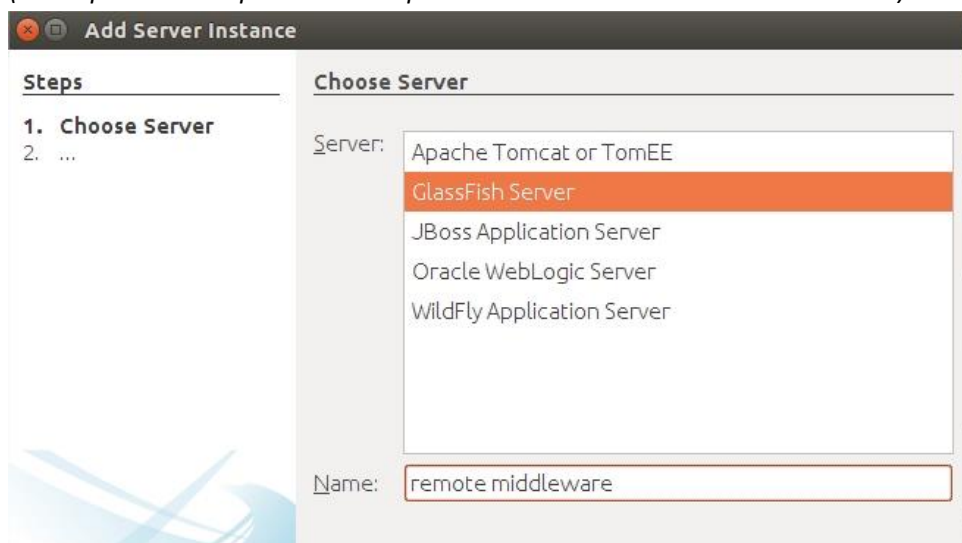
Onglet Services > nœud Servers > Add Server... > Sélectionnez GlassFish Server > Nommez l'instance remote :

Pour le domaine store distant : **remote store**.

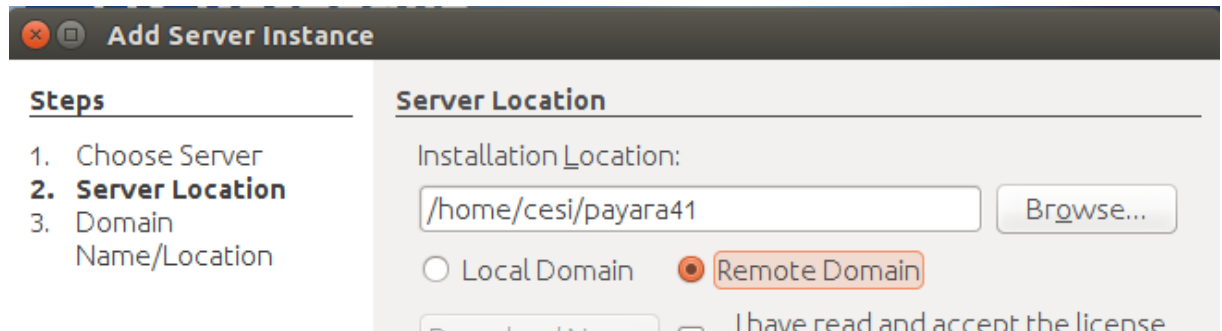
Pour le domaine middleware distant : **remote middleware**.

Pour le domaine bank distant : **remote bank**.

(Les copies d'écran pour cette étape sont relatives au domaine middleware)



Après avoir cliqué sur Next>, cochez sur **Remote Domain** puis passez à l'écran suivant en cliquant sur Next > :



Add Server Instance

Steps

1. Choose Server
- 2. Server Location**
3. Domain Name/Location

Server Location

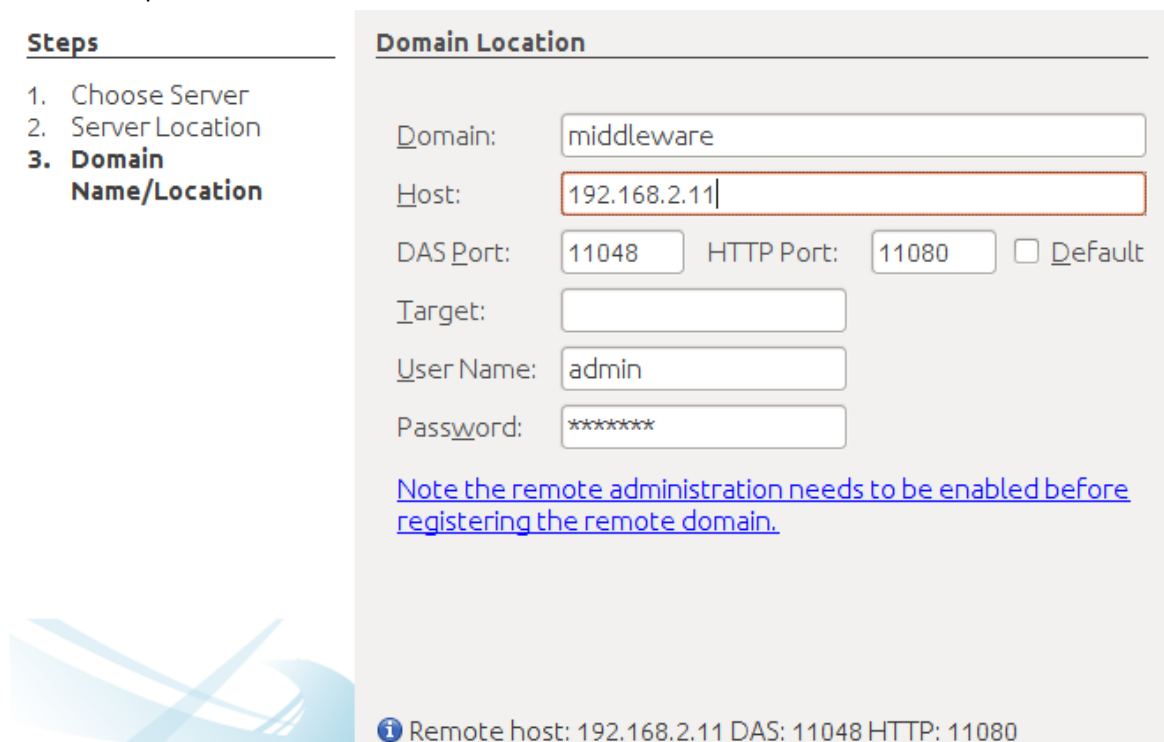
Installation Location:

☐ Local Domain ☒ Remote Domain

☐ I have read and accept the license

Saisissez ensuite les informations relatives au domaine distant :

- Domain : nom du domaine distant (store, middleware ou bank).
- Host : adresse IP de la machine hôte ou nom de la machine hôte si le fichier hosts est correctement configuré.
- Décochez Default
- DAS Port : port administratif (XXX48).
- HTTP Port : port d'écoute de l'instance (XXX80).
- Comme vous ne déployez pas votre projet dans un cluster, il n'y a pas d'IP à saisir dans Target.
- User Name : admin
- Password : le mot de passe admin pour le domaine.
- Cliquez sur Finish.



Steps

1. Choose Server
2. Server Location
- 3. Domain Name/Location**

Domain Location

Domain:

Host:

DAS Port: HTTP Port: ☐ Default

Target:

User Name:

Password:

[Note the remote administration needs to be enabled before registering the remote domain.](#)

Remote host: 192.168.2.11 DAS: 11048 HTTP: 11080

ou

En résumé pour les 3 domaines distants :

	remote store	remote middleware	Remote bank
Domain	store	middleware	bank
Host	192.168.2.10	192.168.2.11	192.168.2.12
DAS	10048	11048	12048
HTTP	10080	11080	12080
Target	192.168.2.10	192.168.2.11	192.168.2.12
User Name	admin	admin	admin
Password	astore	amiddle	abank

Une fois la référence créée, vous devrez sûrement cliquer sur **Refresh** pour que l'icône « Démarré » apparaisse si le domaine distant est démarré.

126. Sur le poste prototype toujours, depuis NetBeans changez les domaines dans lesquels les projets sont déployés.

- Commencez par associer les projets bankFacade-ejb et bankFacade-war au domaine « **remote middleware** » :

Clic <droite> sur le projet bankFacade-[ejb/war] > Properties > Run > Sélectionnez remote middleware > cliquez sur OK pour valider.

Cette opération sur les sous projets vous permettra de tester les services Web déployés dans le domaine middleware distant.

- Associez bankBusiness-ejb à « **remote bank** ».
- Associez webstore à « **remote store** ».

127. Cliquez sur **Run** pour déployer bankFacade-ejb et bankFacade-war dans le domaine middleware distant. NetBeans va vous demander de sélectionner un port pour le débbuging : cliquez sur **OK** (vous pouvez aussi cocher « Don't ask me again »).

Depuis NetBeans, testez le web service SOAP comme vous l'avez fait en amont. Profitez-en pour noter l'adresse du WSDL :

(<http://192.168.13.11:11080/BankingService/BankingServiceBean?WSDL>

Ou

<http://frontsrv:11080/BankingService/BankingServiceBean?WSDL>).

Testez aussi avec cURL votre service REST

```
curl -v -X POST -H "Content-type: application/json" -d '{"ccNumber":
"4569917855","amount": 130}' http://frontSRV:11080/bankFacade-
war/banking/payment
```

Vous pouvez aussi utiliser l'IP : 192.168.2.11

Puis ouvrez la console web d'administration du domaine distant **middleware** et assurez-vous qu'un ou des messages sont placés dans la queue. Eventuellement, ouvrez aussi la console web du domaine distant **bank** pour valider que la queue physPaymentQueue contient bien le message.

128. Déployez le projet bankBusiness-ejb dans le domaine distant **bank**. Dans la console de sortie NetBeans pour « remote bank » vous devriez voir l'affichage correspondant au traitement du paiement.

Il se peut parfois que les logs d'un serveur distant ne s'affichent pas dans la console de sortie NetBeans. Dans ce cas ouvrez la console d'administration web de bank pour visualiser les logs : *Server (Admin Server) > General > view Log Files* :

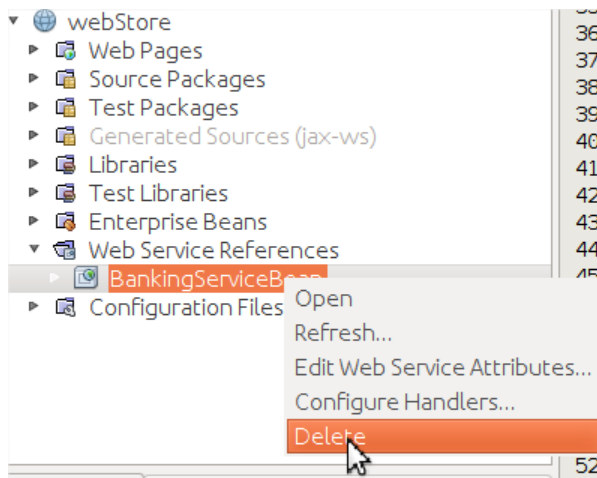
Log Viewer Results (40)			
Records before 333		Log File Record Numbers 333 through 372	
		Records after 372	↑↓
Record Number	Log Level	Message	
372	INFO	l'ordre de paiement <?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="2"><status>V... (details)	
371	INFO	<?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="2"><status>VALIDATED</status><cc... (details)	
370	INFO	[traitement long d'integration dans le processus bancaire de](details)	
369	INFO	<?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="2"><status>VALIDATED</status><cc... (details)	
368	INFO	[traitement long d'integration dans le processus bancaire de](details)	
367	INFO	<?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="2"><status>VALIDATED</status><cc... (details)	
366	INFO	[traitement long d'integration dans le processus bancaire de](details)	
365	INFO	<?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="2"><status>VALIDATED</status><cc... (details)	
364	INFO	[traitement long d'integration dans le processus bancaire de](details)	
363	INFO	<?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="2"><status>VALIDATED</status><cc... (details)	
362	INFO	[traitement long d'integration dans le processus bancaire de](details)	
361	INFO	<?xml version="1.0" encoding="UTF-8" standalone="yes"?><payment id="2"><status>VALIDATED</status><cc... (details)	
360	INFO	[traitement long d'integration dans le processus bancaire de](details)	

129. Dans le projet webStore, vous devez mettre à jour les adresses des services Web SOAP et RESTful pour pointer vers le domaine distant middleware.

- Référence au service Web SOAP :

Supprimez la référence cliente au web service s'exécutant en local (localhost). Les artefacts générés du packaging sont supprimés.

Projet webStore > nœud Web Service References > clic <droite> BankingServiceBean > Delete :



Puis faites pointer vers le web service hébergé dans le domaine middleware distant (IP : 192.168.2.11 ou nom de la machine : frontSRV). Pour ce faire, reportez-vous à l'étape 48.

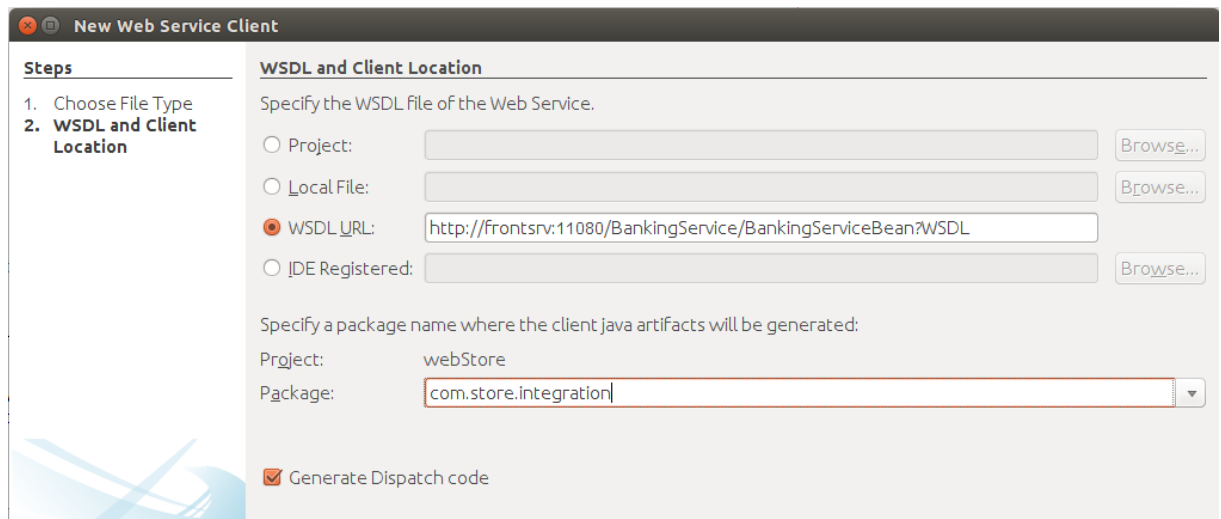
Seule l'adresse du WSDL change par rapport à l'étape 48 :

<http://192.168.2.11:11080/BankingService/BankingServiceBean?WSDL>

Ou

<http://frontsrv:11080/BankingService/BankingServiceBean?WSDL>

N'oubliez-pas de spécifier que les artefacts seront générés dans **com.store.integration**.



- Adresse du service RESTFul :

Dans la méthode **process** de *RestPaymentValidator*, remplacez *localhost* par l'IP du serveur frontSRV 192.168.2.11 ou par le nom de la machine, c'est-à-dire frontSRV. Le port reste inchangé :

```
WebTarget target = client.target("http://191.168.2.11:11080/bankFacade-war/banking/payment");
```

Ou

```
WebTarget target = client.target("http://frontSRV:11080/bankFacade-war/banking/payment");
```

De même dans la classe PaymentOrderBean modifiez la valeur de la variable baseURL en remplaçant localhost par l'IP 192.168.2.11 ou frontSRV :

```
@Named(value = "paymentOrderModel")
@RequestScoped
public class PaymentOrderBean{

    private final String baseURL = "http://frontSRV:11080/bankFacade-war/banking/payment";
    ....
}
```

130. Cliquez sur *[Clean &] Build puis Run* pour lancer projet **webStore**.
Testez votre application distribuée.

Vous avez réussi à répartir votre système sur plusieurs machines !!!

Annexe

Sauvegarde et restauration d'un domaine

On suppose que les domaines localisés seront localisés dans le dossier `/home/cesi/localdomainsBCKP`, `cesi` étant le compte utilisateur.

On prendra pour exemple le domaine `store`.

Eteignez le domaine à sauvegarder :

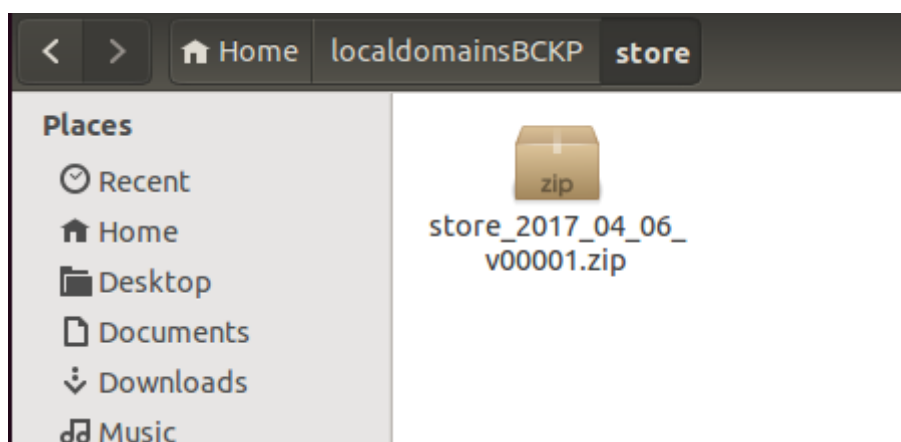
```
asadmin stop-domain store
```

Sauvegardez le domaine `store` dans le dossier `localdomainsBCKP` :

```
asadmin backup-domain --backupdir /home/cesi/localdomainsBCKP store
```

Notez que le chemin du dossier de sauvegarde doit être un chemin absolu.

Le dossier `localdomainsBCKP` contient désormais un dossier `store` contenant un zip contenant la sauvegarde :



Home correspondant au dossier utilisateur (dans notre cas, `/home/cesi`)

Pour restaurer un domaine, il suffit d'exécuter :

```
asadmin restore-domain --backupdir /home/cesi/localdomainsBCKP store
```