

# MODULE 6

## Applications n-tiers

# Les applications n-tiers

## Plan du module “Applications n-tiers”

- Les architectures n-tiers
- La couche Persistance
- La couche Métier
- La couche Présentation
- Pour approfondir

# Pour approfondir

Les web services

# Que sont les web services ?

Web services = applications fournissant des services pour d'autres **applications** et accessibles par **HTTP**

- il ne s'agit pas de fonctions utilisables directement par un humain

# Que sont les web services ?

Un web service peut être déployé dans un conteneur web ou dans un conteneur d'EJB.

# Que sont les web services ?

Un web service fournit un ensemble de services appelés **opérations**.

Deux types de web service :

- RPC-like → JAX-WS
- REST → JAX-RS

# Pour approfondir

Les web services de type RPC

# Web services façon JAX-WS

JAX-WS : Java API for XML-based Web Services



# Web services façon JAX-WS

JAX-WS : web services de type **RPC**

- mode client / serveur
  - requêtes / réponses en XML
- simulation d'un appel de procédure
- protocole : **SOAP** (Simple Object Access Protocol)

# Exemple de messages SOAP

WildFly Manage x Method invocati x http://localhost: x Getting Started x Coupe du Monde x Method invocati x SOAP - Wikipedi x

localhost:8080/BrokerWS/BrokerWS?Tester

Applications Google Y Mail Gmail IB Y Fin Bourse Arduino W Unicode char Concepteur de jeux Collège EDT IUT YT Autres favoris

## getQuote Method invocation

Method parameter(s)

Type	Value
java.lang.String	MacDonald

Method returned

java.lang.Float : "57.5"

SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <S:Body xmlns:ns2="http://ws.broker/">
    <ns2:getQuote>
      <stockName>MacDonald</stockName>
    </ns2:getQuote>
  </S:Body>
</S:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <S:Body xmlns:ns2="http://ws.broker/">
    <ns2:getQuoteResponse>
      <return>57.5</return>
    </ns2:getQuoteResponse>
  </S:Body>
</S:Envelope>
```

# Web services façon JAX-WS

SOAP utilise une seule méthode HTTP :  
POST

- car POST permet d'attacher des données à la requête HTTP

⇒ HTTP est utilisé comme protocole de **transport**

# Exemple de requête SOAP

POST /InStock HTTP/1.1

Host: www.example.org

Content-Type: application/soap+xml;  
charset=utf-8

Content-Length: 299

SOAPAction: "http://www.w3.org/2003/05/soap-  
envelope"

# Exemple de requête SOAP

```
<?xml version="1.0"?>
```

```
<soap:Envelope
```

```
xmlns:soap="http://www.w3.org/2003/05/soap-  
envelope"
```

```
xmlns:m="http://www.example.org/stock/Reddy">
```

```
  <soap:Header>
```

```
  </soap:Header>
```

# Exemple de requête SOAP

```
<soap:Body>
```

```
  <m:GetStockPrice>
```

```
    <m:StockName>GOOG</m:StockName>
```

```
  </m:GetStockPrice>
```

```
</soap:Body>
```

```
</soap:Envelope>
```

# Découverte d'un web service

La **localisation** d'un service peut être faite au travers d'un annuaire **UDDI** (Universal Description Discovery and Integration)

# Description d'un web service

La **description** des services fournis par un web service est faite au travers d'un fichier **WSDL** (Web Service Definition Language).

Un client découvre le détail de la communication avec un web service par lecture de son fichier WSDL.



# Format de fichier XML

JAX-WS utilise largement XML :

- messages SOAP
- fichier WSDL

Description du format d'un fichier XML :  
**XSD** (XML Schema Description)

# Architecture orientée services

## **SOA** : Service Oriented Architecture

### Principes :

- définition de services adhoc  $\Rightarrow$  SOAP
- description des services  $\Rightarrow$  WSDL
- découverte des services  $\Rightarrow$  UDDI
- format de fichier XML  $\Rightarrow$  XSD

# Exemple de client

```
CalculatorWS_Service service = new  
CalculatorWS_Service();
```

```
CalculatorWS port =  
service.getCalculatorWSPort();
```

```
int result = port.add(4, 8);
```

```
System.out.println("Result = " + result);
```

```
result = port.multiply(10, 3);
```

```
System.out.println("Result = " + result);
```



Web service

# Exemple de client

```
CalculatorWS_Service service = new  
CalculatorWS_Service();
```

```
CalculatorWS port =  
service.getCalculatorWSPort();
```

“Port” du  
web service

```
int result = port.add(4, 8);
```

```
System.out.println("Result = " + result);
```

```
result = port.multiply(10, 3);
```

```
System.out.println("Result = " + result);
```

# Exemple de client

```
CalculatorWS_Service service = new  
CalculatorWS_Service();
```

```
CalculatorWS port =  
service.getCalculatorWSPort();
```

```
int result = port.add(4, 8);
```

```
System.out.println("Result = " + result);
```

```
result = port.multiply(10, 3);
```

```
System.out.println("Result = " + result);
```

Utilisation du  
web service

# Exemple de client

```
CalculatorWS_Service service = new  
CalculatorWS_Service();  
  
CalculatorWS port =  
service.getCalculatorWSPort();  
  
int result = port.add(4, 8);  
  
System.out.println("Result = " + result);  
  
result = port.multiply(10, 3);  
  
System.out.println("Result = " + result);
```

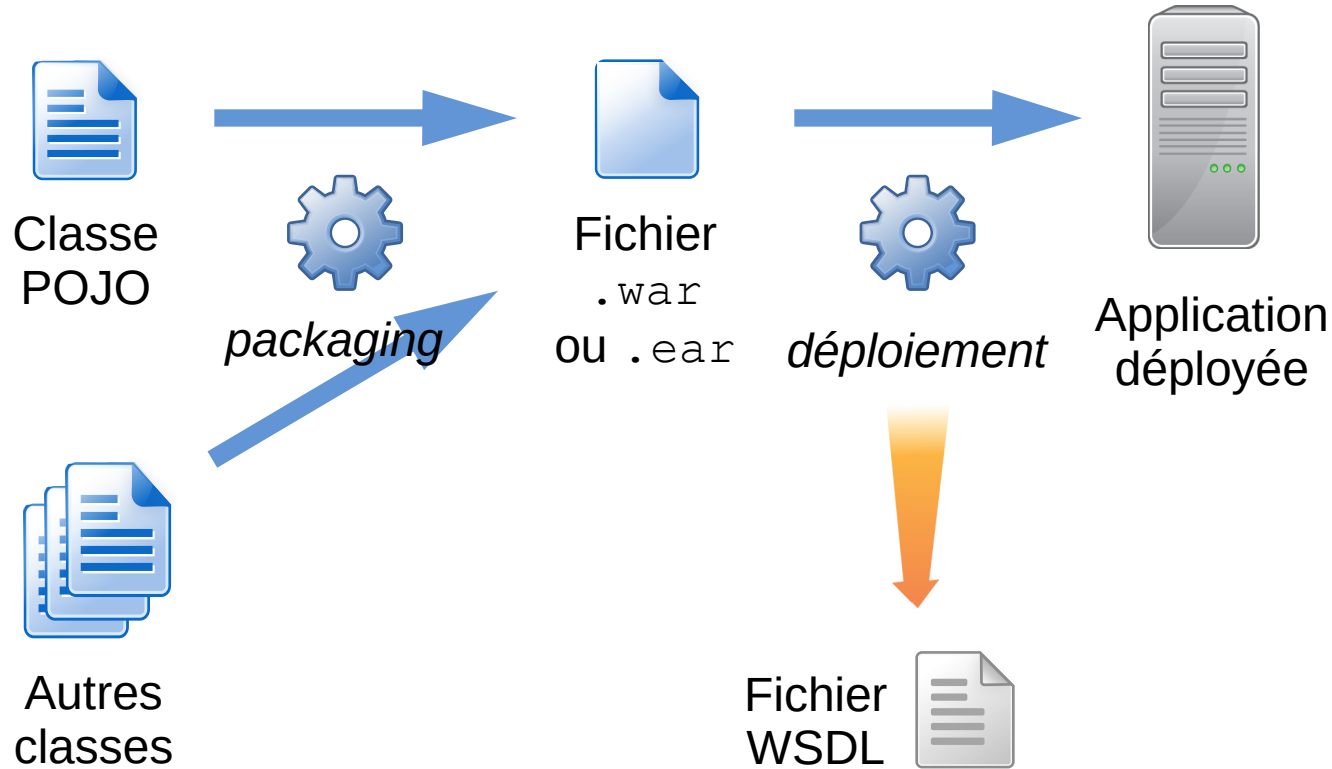
Utilisation du  
web service

# Développement d'un web service

Développement « from scratch » ou à partir d'un fichier WSDL :

- “From scratch” : définition d'un POJO et ajout d'annotations par l'IDE
- Fichier WSDL : l'IDE analyse le fichier WSDL et génère un POJO annoté prêt à être complété

# Développement d'un web service



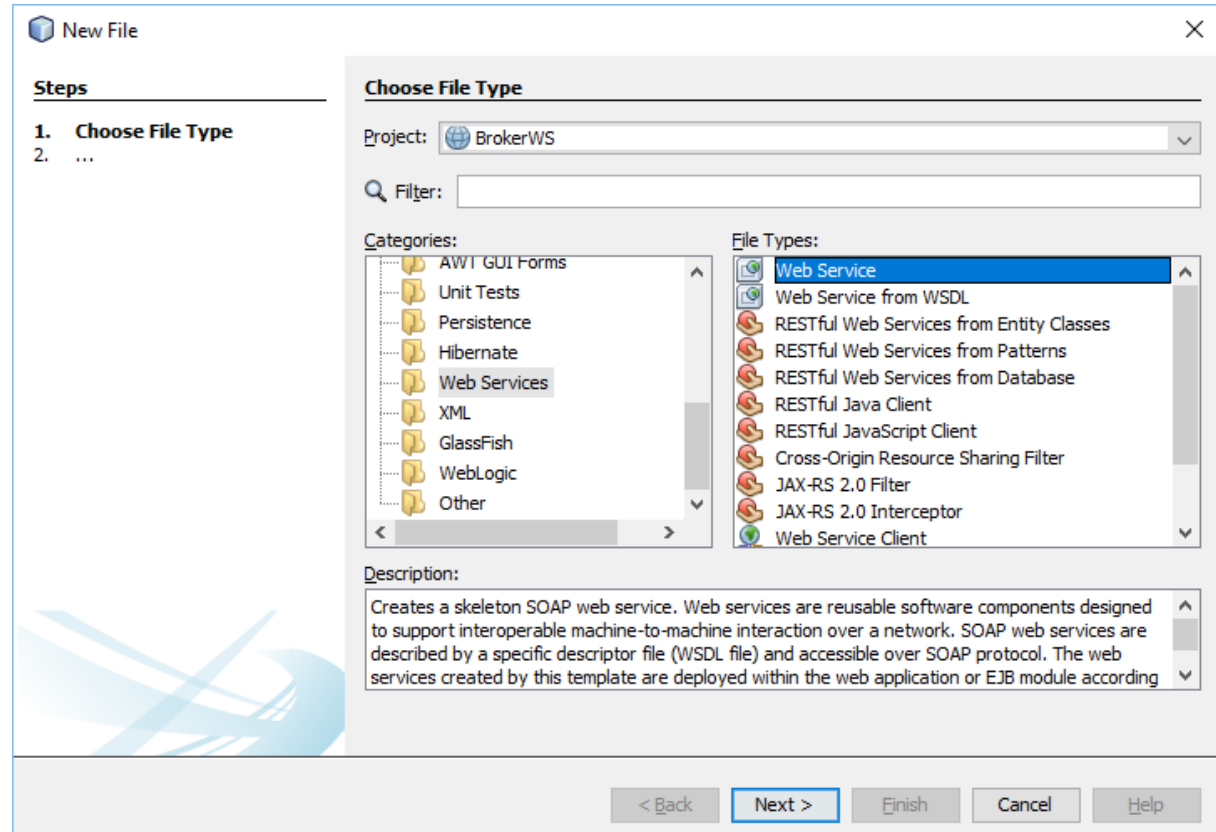


# Utilisation des annotations

```
@WebService(serviceName = "CalculatorWS")  
public class CalculatorWS {  
    @WebMethod(operationName = "add")  
    public int add(@WebParam(name="i") int i,  
                  @WebParam(name="j") int j) {  
        return i + j;  
    }  
}
```

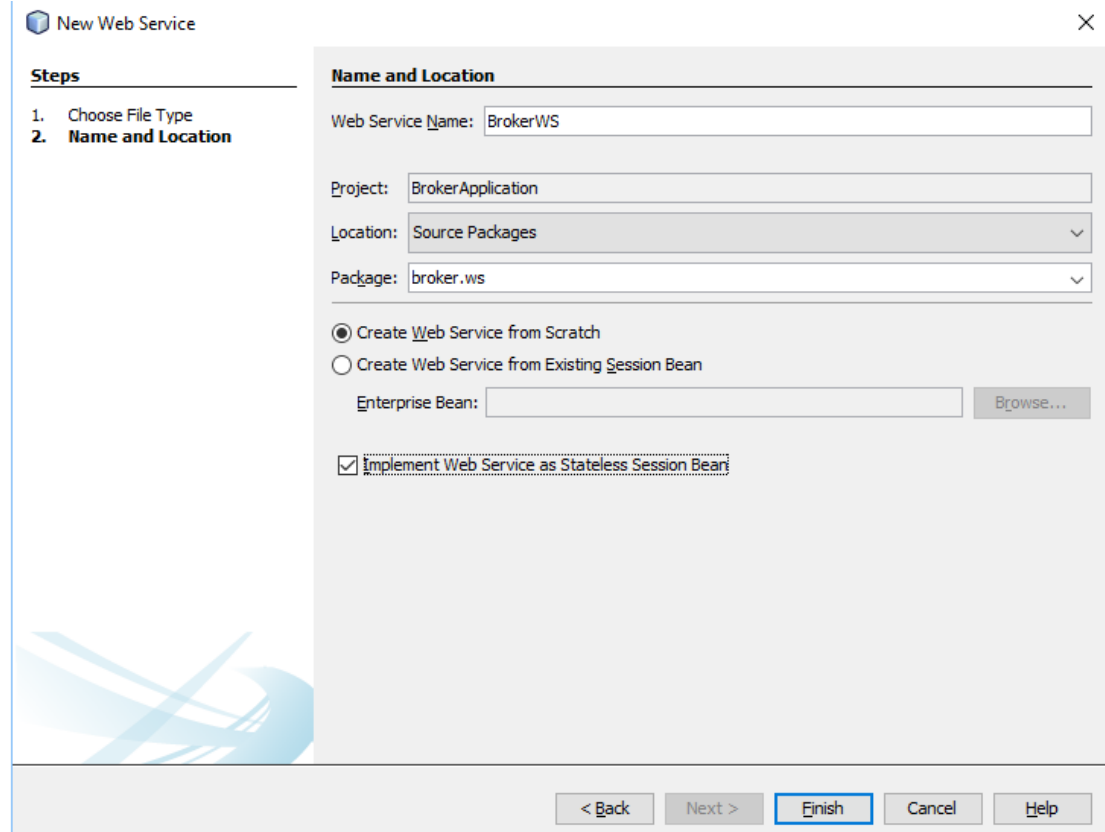
# Développement d'un web service

## Création d'un web service



# Développement d'un web service

Création d'un  
web service basé  
sur un EJB  
Session  
stateless

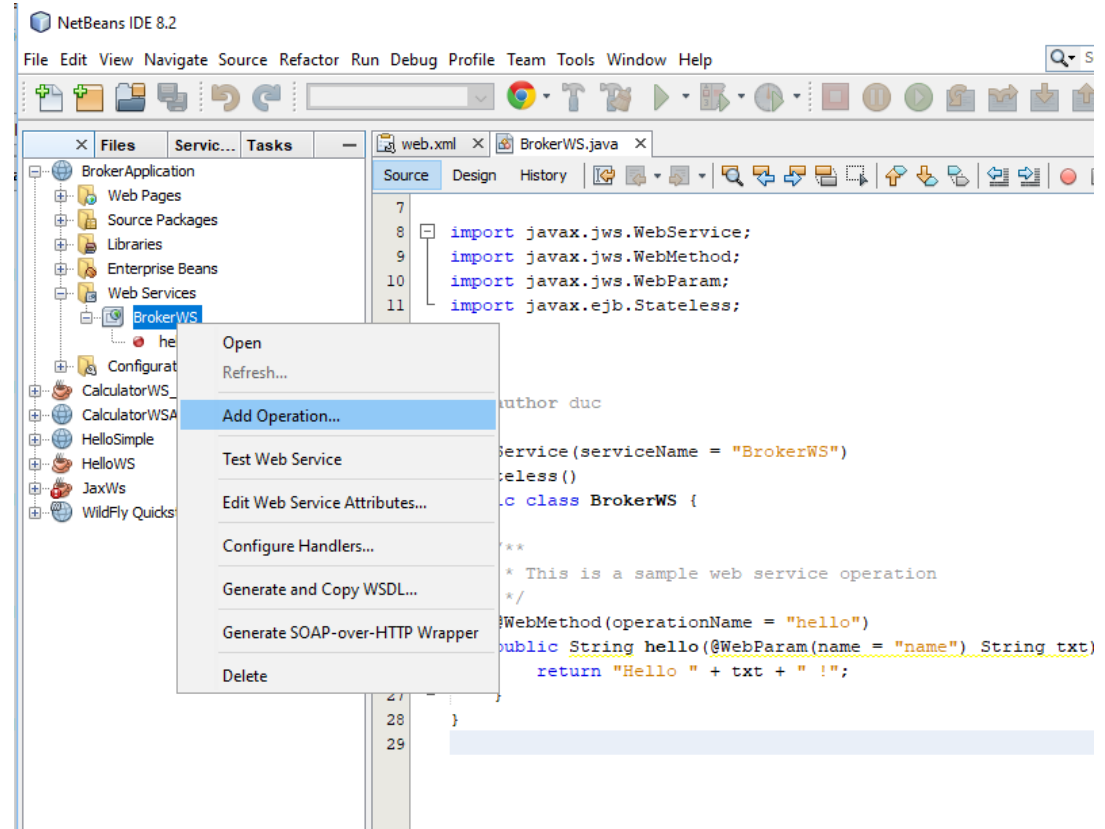


The screenshot shows the 'New Web Service' wizard with the following configuration:

- Steps:**
  1. Choose File Type
  2. **Name and Location**
- Name and Location:**
  - Web Service Name: BrokerWS
  - Project: BrokerApplication
  - Location: Source Packages
  - Package: broker.ws
- Options:**
  - ☒ Create Web Service from Scratch
  - ☐ Create Web Service from Existing Session Bean
    - Enterprise Bean: [Empty field] [Browse...]
  - ☒ Implement Web Service as Stateless Session Bean
- Navigation:** < Back, Next >, **Finish**, Cancel, Help

# Développement d'un web service

Ajout d'une  
opération sur le  
web service



# Développement d'un web service

Ajout d'une  
opération sur le  
web service :  
saisie du type de  
retour et du type  
des paramètres

**Add Operation**

Name:

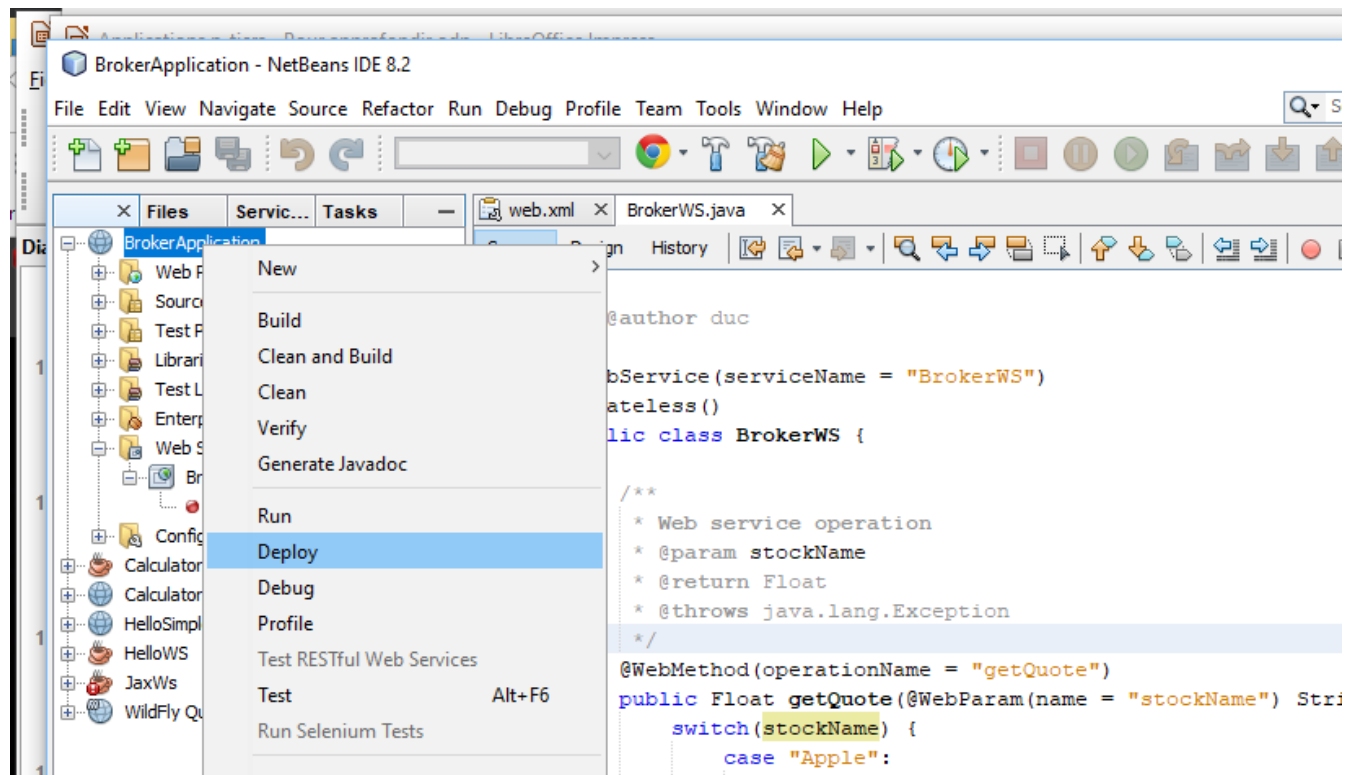
Return Type:

Parameters  Exceptions

Name	Type	Final
stockName	java.lang.String	<input type="checkbox"/>

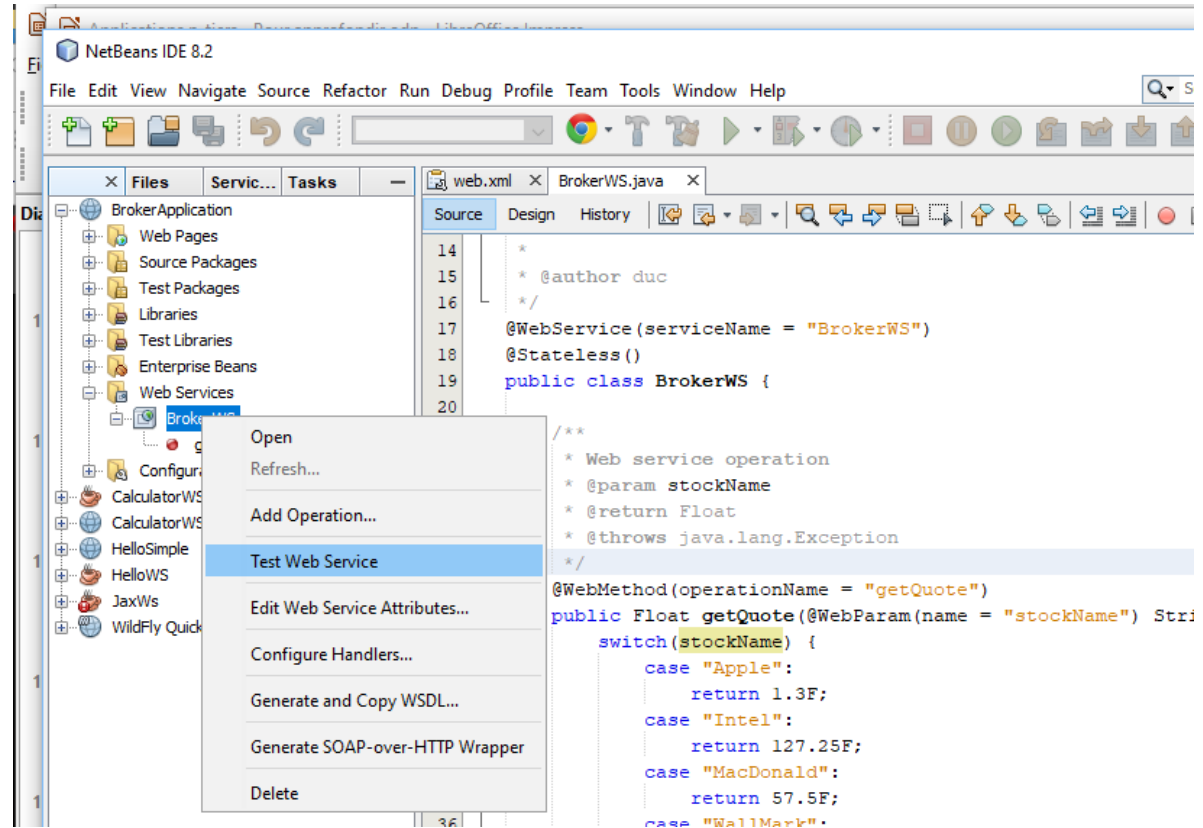
# Déploiement d'un web service

Déploiement  
du web service  
sur le serveur  
d'application



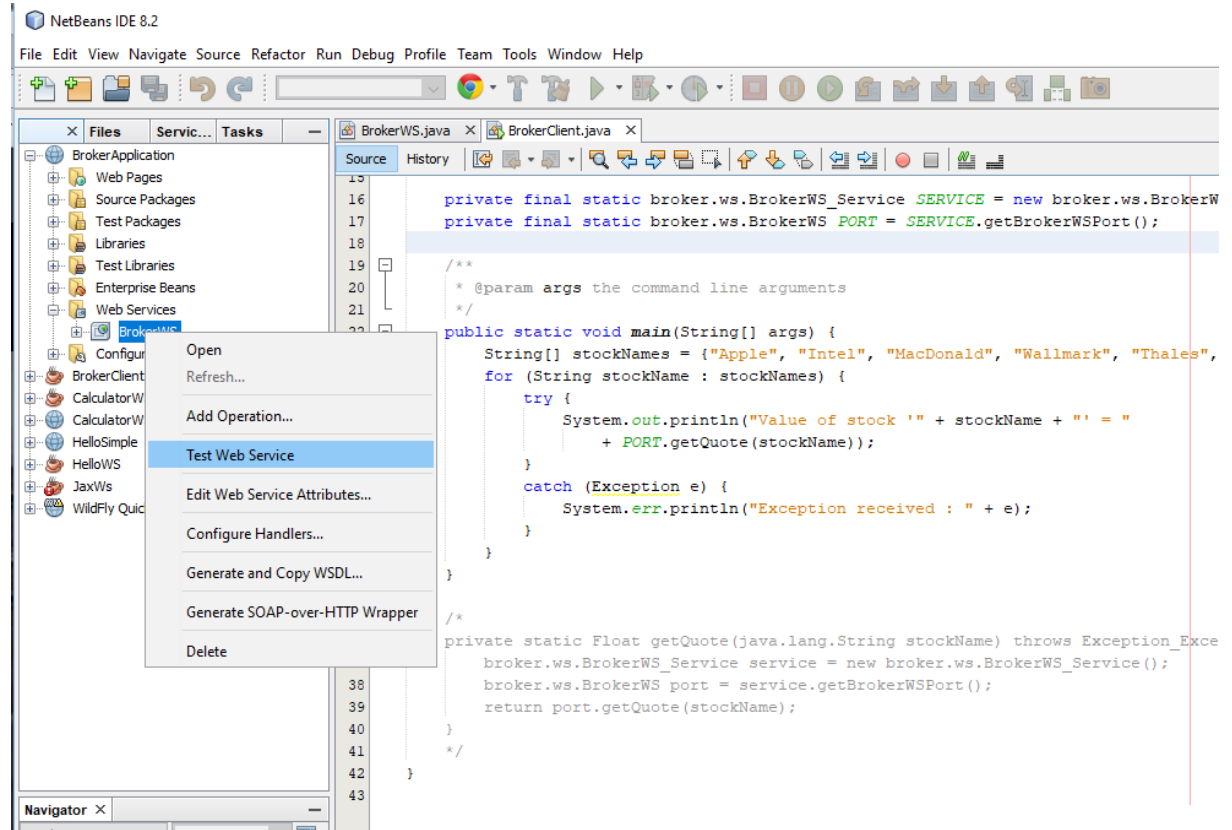
# Test d'un web service

Test du web service via le serveur d'application



# Test d'un web service

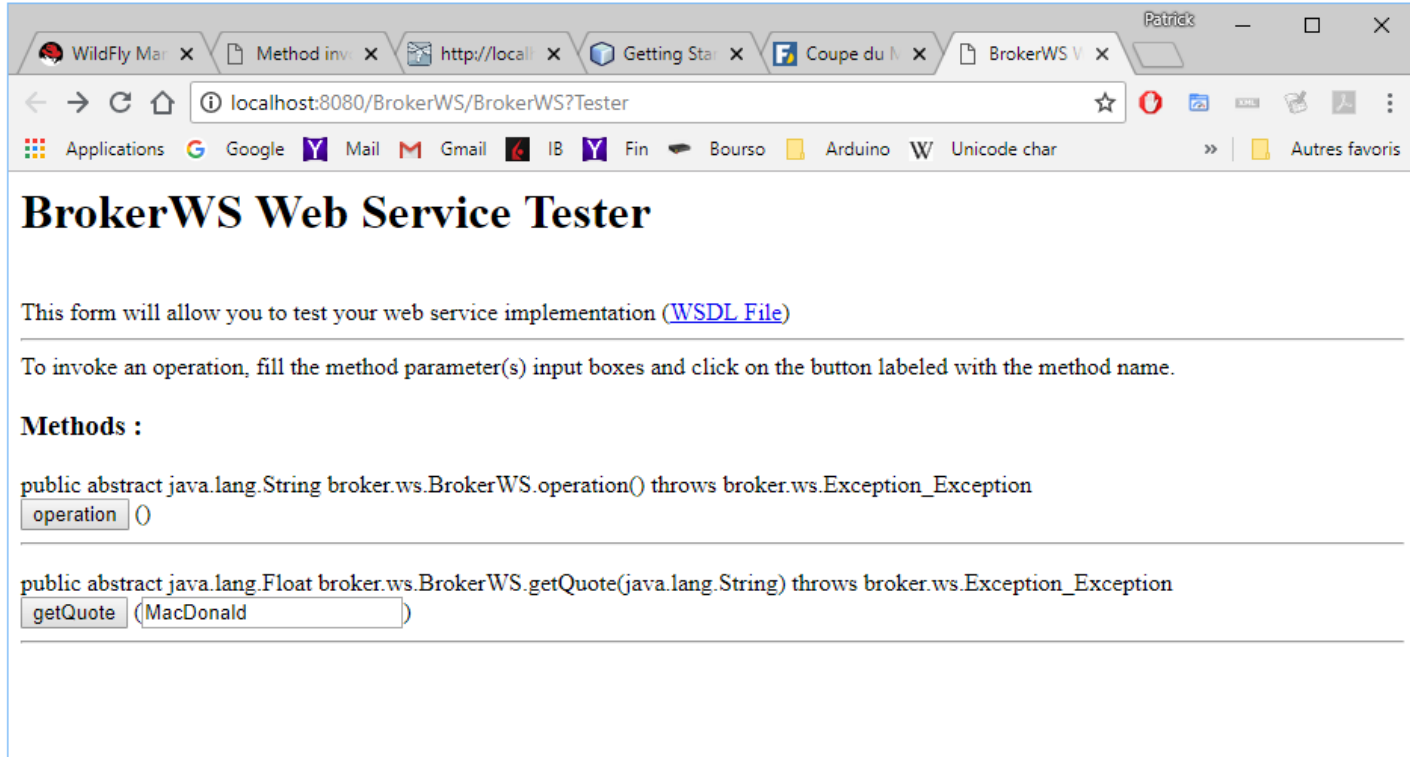
GlassFish fournit un service de test de web services intégré dans NetBeans.





# Test d'un web service

Démarrage  
d'une page  
web  
permettant  
de tester  
directement  
le service  
web.



The screenshot shows a web browser window with the title "Patrick". The address bar displays "localhost:8080/BrokerWS/BrokerWS?Tester". The browser's tab bar shows several open tabs: "WildFly Mar", "Method inv", "http://local", "Getting Sta", "Coupe du N", and "BrokerWS V". The browser's toolbar includes icons for back, forward, refresh, and home, along with a search bar and a star icon. The browser's address bar shows "localhost:8080/BrokerWS/BrokerWS?Tester". The browser's toolbar includes icons for Applications, Google, Mail, Gmail, IB, Fin, Bours, Arduino, and Unicode char, along with a search bar and a star icon. The browser's address bar shows "localhost:8080/BrokerWS/BrokerWS?Tester".

## BrokerWS Web Service Tester

This form will allow you to test your web service implementation ([WSDL File](#))

To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.

**Methods :**

`public abstract java.lang.String broker.ws.BrokerWS.operation() throws broker.ws.Exception_Exception`

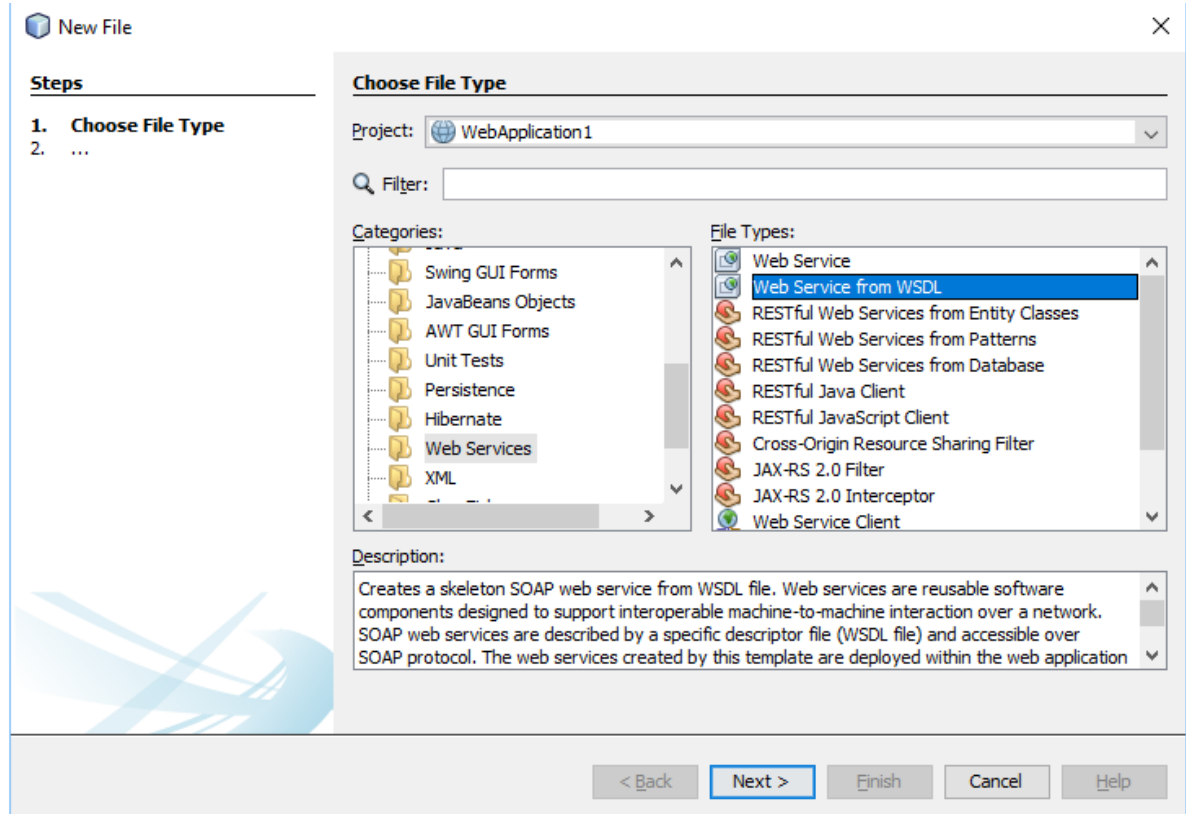
`public abstract java.lang.Float broker.ws.BrokerWS.getQuote(java.lang.String) throws broker.ws.Exception_Exception`

# Développement d'un web service

Deuxième possibilité : développement à partir du fichier WSDL

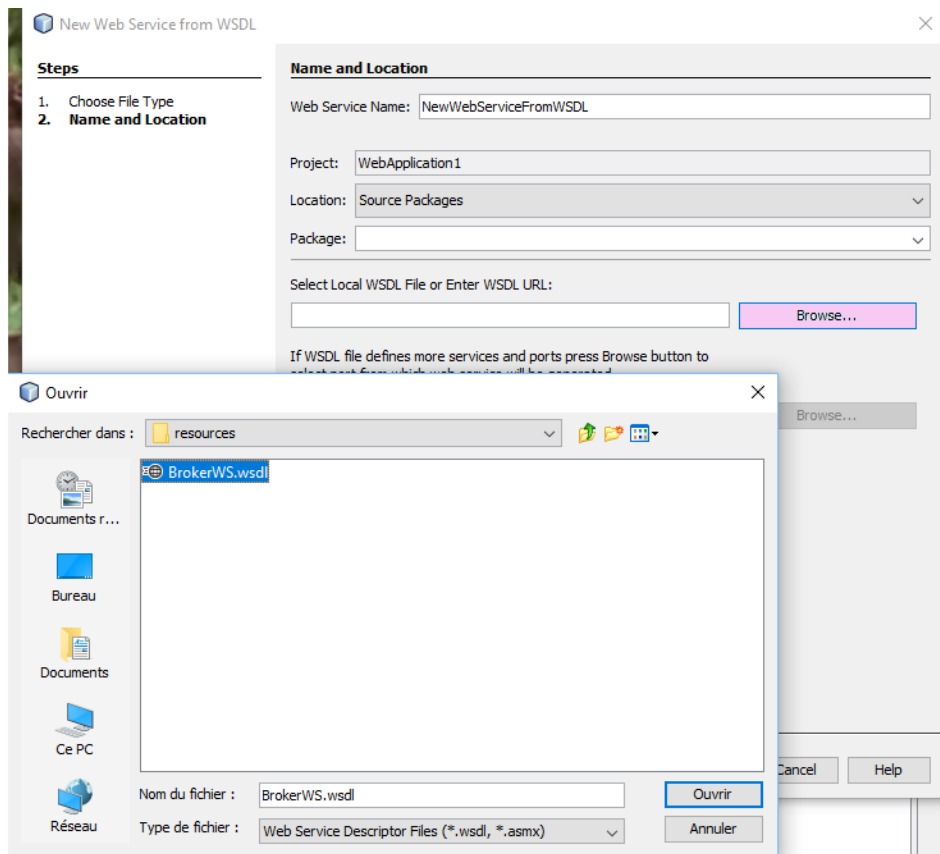
# Développement d'un web service

Développement  
d'un web service  
à partir d'un  
fichier WSDL  
(étape 1)



# Développement d'un web service

Développement  
d'un web service  
à partir d'un  
fichier WSDL  
(étape 2)

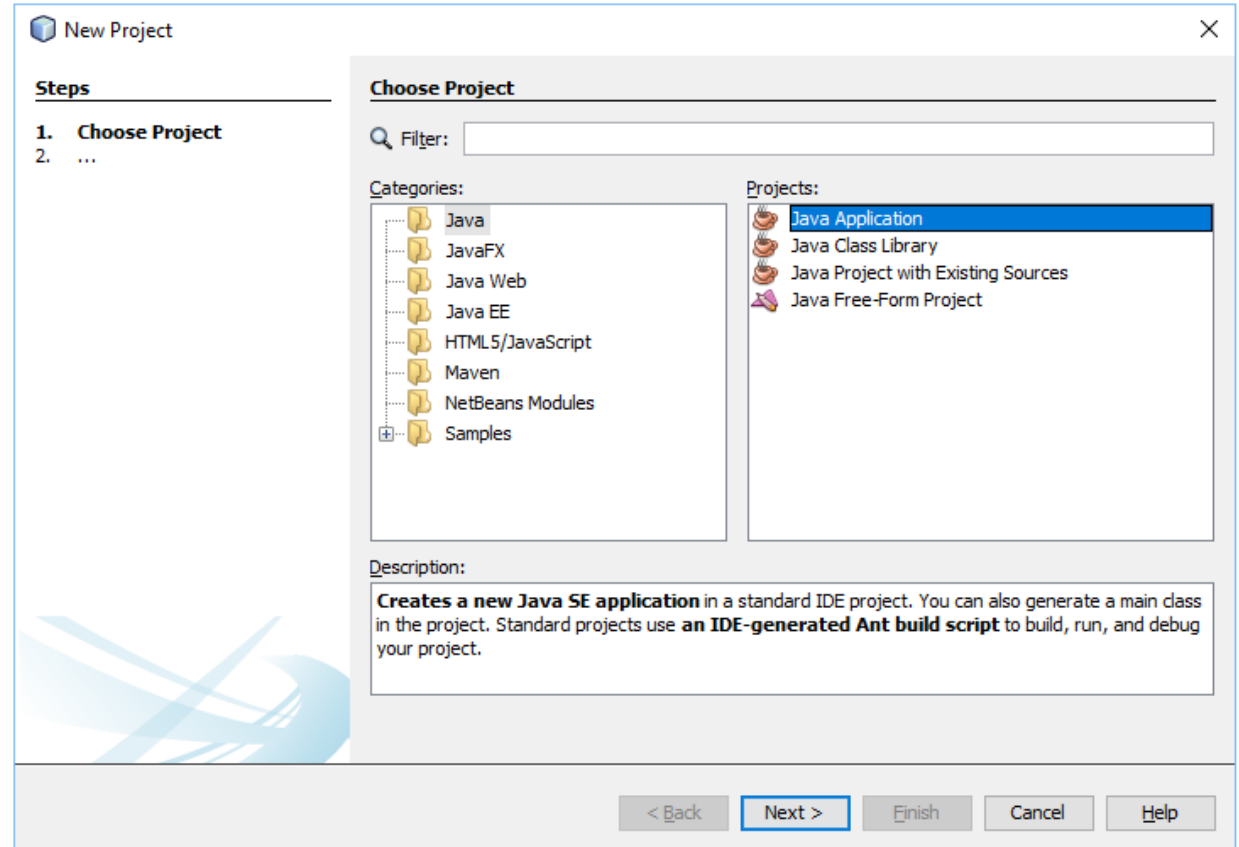


# Développement d'un client web service

Un client de web service peut être une simple application Java, une servlet, une JSP, un EJB...

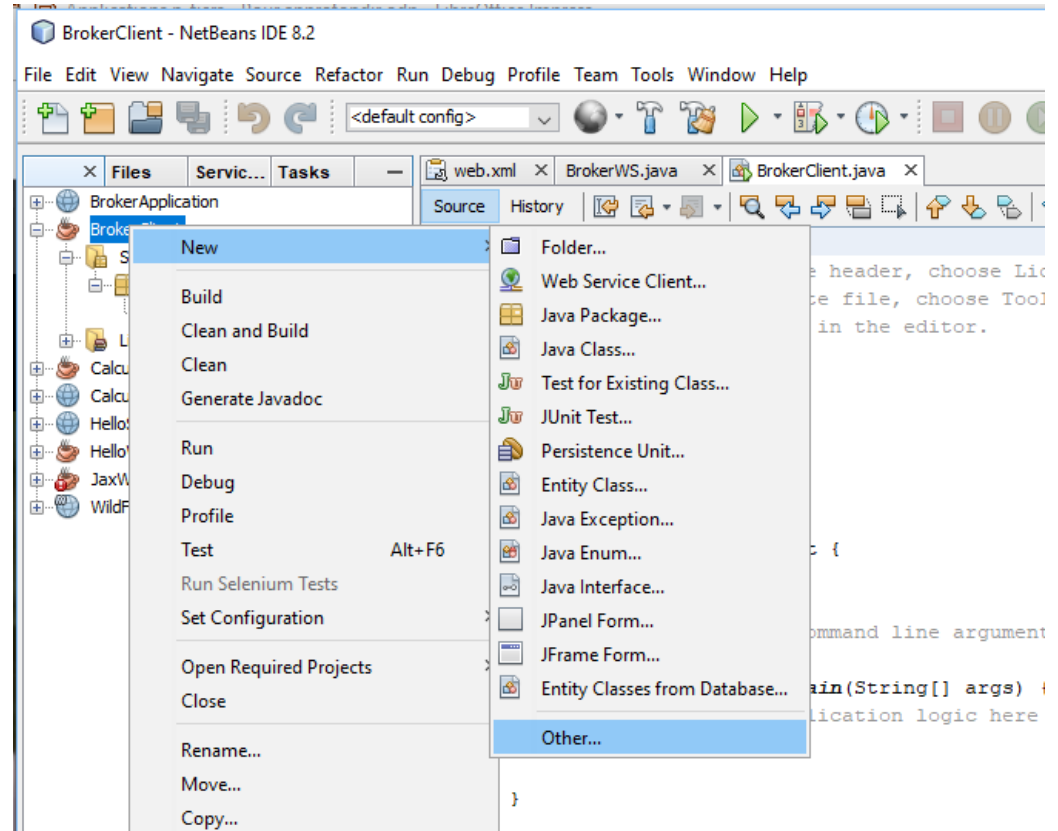
# Développement d'un client web service

## Création d'une simple application Java



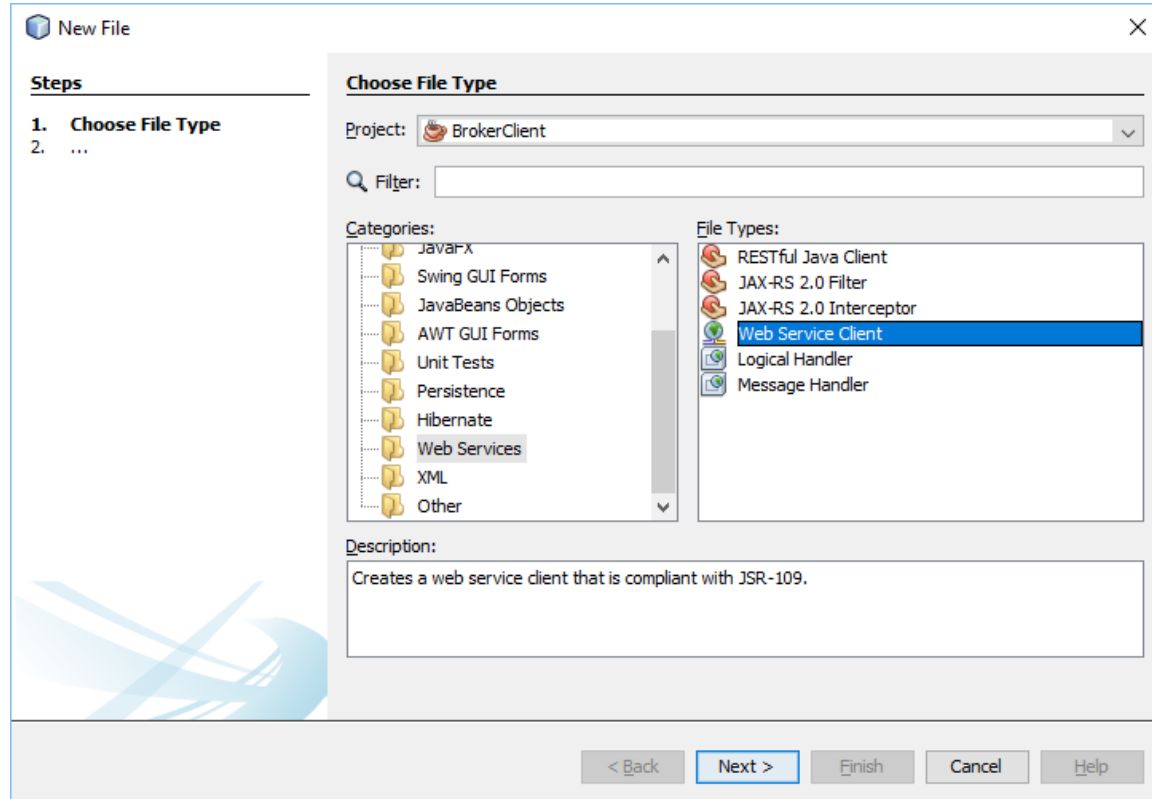
# Développement d'un client web service

## Création du client web service (étape 1)



# Développement d'un client web service

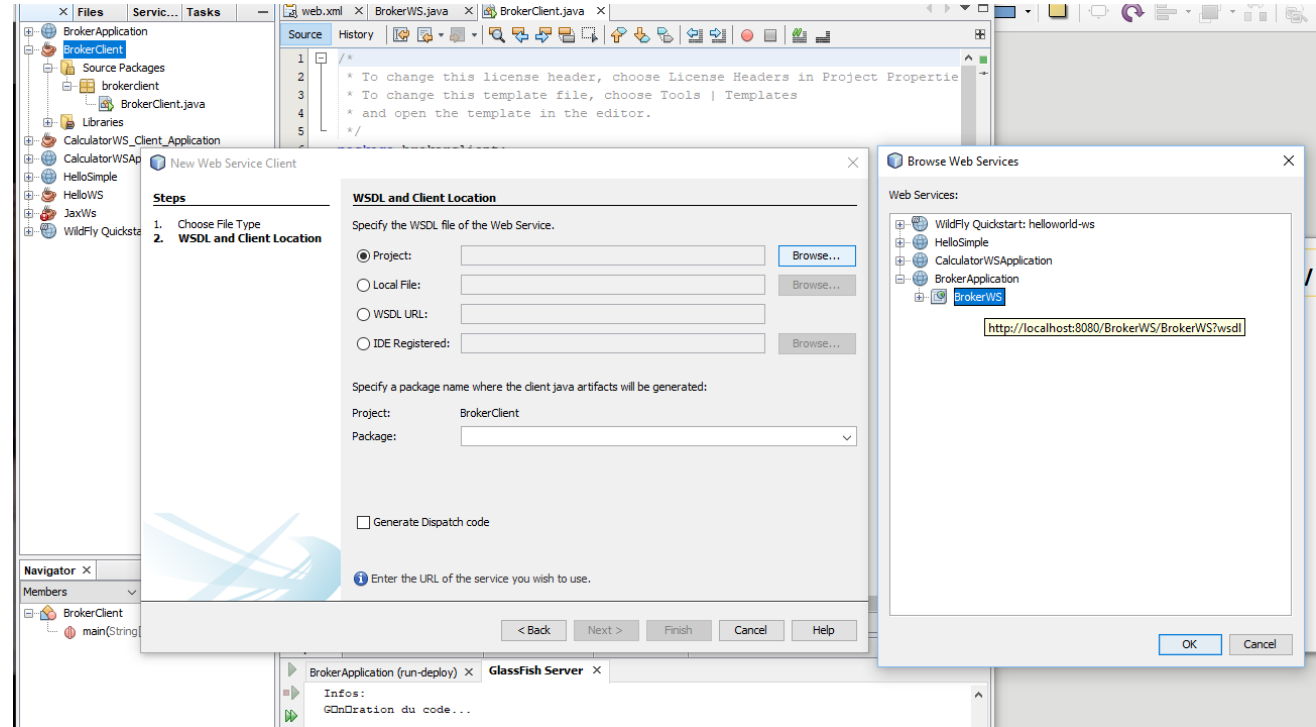
## Création du client web service (étape 2)





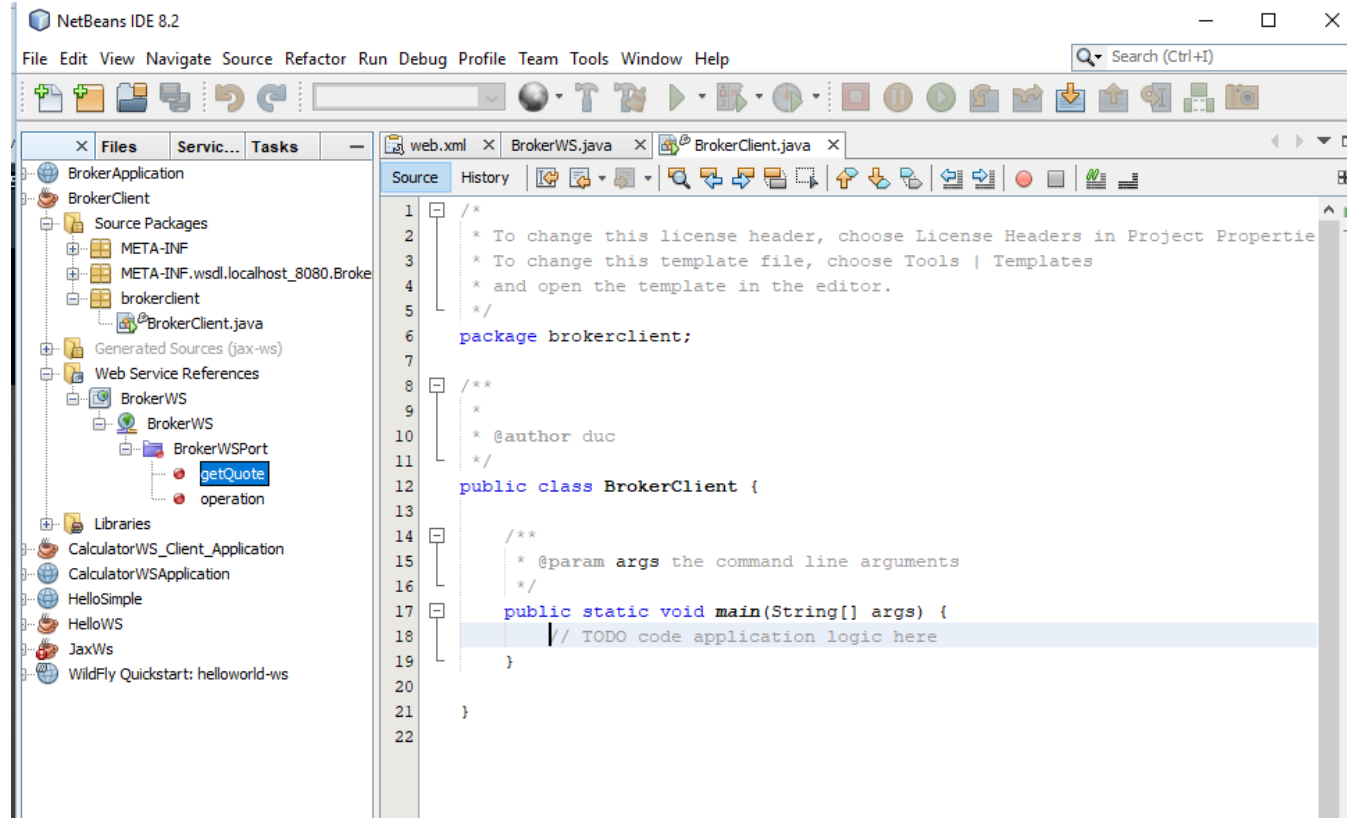
# Développement d'un client web service

Création du client web service  
(étape 3 : sélection du web service utilisé)



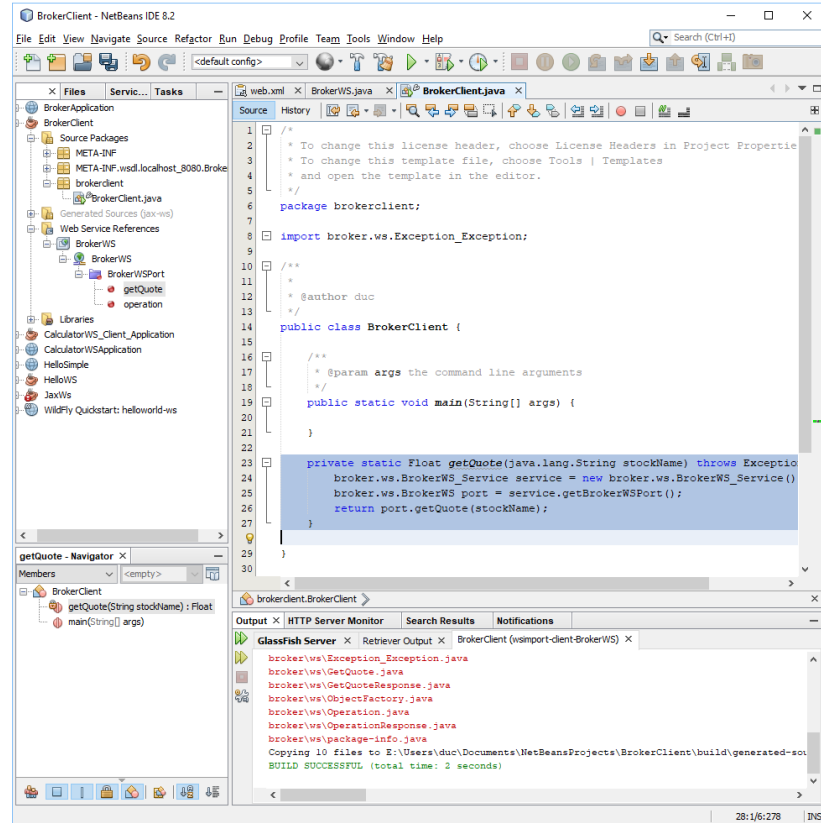
# Développement d'un client web service

Création du client web service (étape 4 : drap & drop de l'opération à invoquer)



# Développement d'un client web service

Création du client web service (étape 4 : résultat du drag & drop)



# Extrait du fichier WSDL

...

```
<portType name="CalculatorWS">
```

```
  <operation name="add">
```

```
    <input message="add" />
```

```
    <output message="addResponse" />
```

```
  </operation>
```

```
</portType>
```

...

# Extrait du fichier WSDL

...

```
<service name="BrokerWS">
```

```
  <port name="BrokerWSPort"  
binding="tns:BrokerWSPortBinding">
```

```
    <soap:address  
location="http://localhost:8080/WebApplication1/Bro  
kerWS" />
```


```
  </port>
```

```
</service>
```

...

# Utilisation des annotations

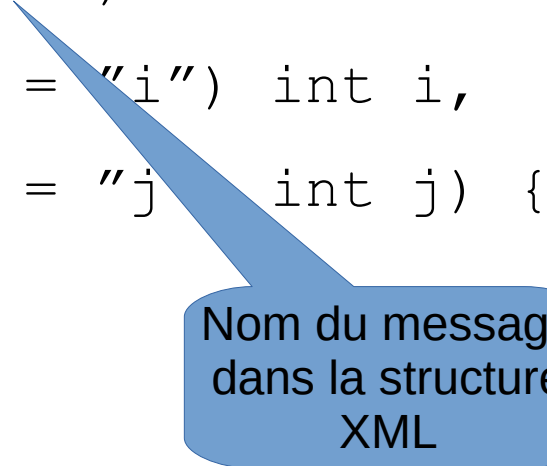
```
@WebService(serviceName = "CalculatorWS")
public class CalculatorWS {
    @WebMethod(operationName = "add")
    public int add(@WebParam(name = "i") int i,
                  @WebParam(name = "j") int j) {
        return i + j;
    }
}
```



Nom du  
web service  
dans l'URL

# Utilisation des annotations

```
@WebService(serviceName = "CalculatorWS")
public class CalculatorWS {
    @WebMethod(operationName = "add")
    public int add(@WebParam(name = "i") int i,
                  @WebParam(name = "j") int j) {
        return i + j;
    }
}
```



Nom du message  
dans la structure  
XML

# Web services façon JAX-WS : exercice

Réaliser l'exercice 14 (service de courtage)



# Pour approfondir

Les web services de type REST

# Web services façon JAX-RS

JAX-RS : Java API for RESTful Web Services

Alors, qu'est-ce qu'un web service RESTful ?

# Web services façon JAX-RS

REST = REpresentational State Transfer

Ce n'est :

- ni un protocole
- ni un standard
- ni un ensemble d'outils

# Web services façon JAX-RS

REST est un **style architectural** :

- définition de **ressources** identifiées par URI
- interface uniforme (opérations **C R U D**)
- échange de **représentations** d'une ressource
- pas d'enregistrement d'**état conversationnel** sur le serveur
- utilisation de **liens** pour indiquer les changements d'état des ressources

# Web services façon JAX-RS

REST est conceptuellement indépendant de **HTTP**...

...mais en pratique, HTTP est toujours utilisé comme **protocole applicatif** entre serveurs et clients REST

# Web services façon JAX-RS

HTTP en tant que **protocole applicatif** :

- verbes HTTP avec **sémantique précise**
- codes de retour HTTP avec **sémantique précise**
- structure des URI **significative**

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- GET : utilisé pour **récupérer** une représentation d'une ressource associée à l'URI passée en argument du GET

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- GET :

- format de représentation **négociable**  
au travers de **directives HTTP**  
(Accept) : texte, HTML, XML, JSON,  
binaire, PDF, ...



# Concept d'interface uniforme

Les verbes HTTP utilisés :

- GET :

- le serveur retourne normalement le code 200 ainsi qu'une représentation de la ressource associée à l'URI mentionnée dans la requête GET

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- GET : exemple (lire détails utilisateur)

- Requête :

HTTP 1.1

GET /utilisateurs/duc

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- GET :

- Réponse :

```
HTTP 1.1 200 OK
```

```
<h1>Patrick Duc</h1>
```

```
<h2>Mail : duc@any.com<h2>
```

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- POST : utilisé pour créer une ressource en **laissant le serveur choisir** le nom de la ressource
  - exemple d'un post sur un blog : le rédacteur du post n'est **pas** intéressé par **l'identifiant** du post

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- POST : utilisé aussi pour créer un **élément** de ressource, donc sans génération d'une URI
  - exemple d'une annotation sur une ressource

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- POST :

- le serveur répond normalement par un code de retour 201 (`Created`) en indiquant une **URI** qui est l'identifiant de la ressource créée

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- POST :

- Dans le cas où il n'y a pas de ressource créée, le serveur répond normalement par un code de retour 200 (OK) ou 204 (No Content)

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- POST : exemple (nouveau post)

- Requête :

```
HTTP 1.1
```

```
POST /monblog
```

```
bla bla bla...
```



# Concept d'interface uniforme

Les verbes HTTP utilisés :

- POST :

- Réponse :

`HTTP 1.1 201 Created`

`Location: /monblog/738025`

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- PUT : utilisé pour créer une ressource dont l'identifiant est choisi par le client
  - cas de la création d'un nouvel utilisateur d'un système, par exemple

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- PUT : utilisé pour modifier une ressource existante
  - changement du contenu d'une ressource, typiquement

# Concept d'interface uniforme

Les verbes HTTP utilisés :

– PUT :

- sur création d'une ressource, le serveur doit répondre par un code de retour 201 (`Created`) en indiquant une **URI** qui est le **nom** de la ressource

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- PUT :

- sur modification d'une ressource, le serveur doit répondre par un code de retour 200 (OK) ou 204 (No Content)

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- PUT : exemple (nouvelle photo officielle de la FIAT Panda)

- Requête :

```
HTTP 1.1
```

```
PUT /voitures/fiat/panda
```

```
...image JPEG...
```

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- PUT :

- Réponse :

HTTP 1.1 204 No Content

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- DELETE : utilisé pour supprimer une ressource
  - exemple d'un article retiré des ventes sur un site marchand



# Concept d'interface uniforme

Les verbes HTTP utilisés :

– DELETE :

- réponse normale = code de retour 200 (si la réponse contient une information), 202 (en cours de traitement) ou 204 (si la réponse ne contient pas d'information)

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- DELETE : exemple (suppression d'article)
  - Requête :

```
HTTP 1.1
```

```
DELETE /monsite/articles/86553
```

# Concept d'interface uniforme

Les verbes HTTP utilisés :

- DELETE :

- Réponse :

HTTP 1.1 204 No Content

# Avantages de l'interface uniforme

Les équipements réseau ont connaissance du **protocole applicatif** HTTP et peuvent éviter certains traitements inutiles

- cache de données
- relance de requêtes **idempotent**  
PUT / DELETE

# Avantages de l'interface uniforme

Surtout, utiliser des requêtes CRUD signifie que **la sémantique des traitements est toujours la même**, seules les ressources changent

- équivalent à l'utilisation de SQL : les requêtes `INSERT / UPDATE / SELECT / DELETE` gardent leur sémantique, seules les tables changent

# Avantages de l'interface uniforme

Avec JAX-WS et SOAP, la logique des traitements **change** potentiellement avec chaque nouvel ensemble de services

- définition des services adhoc
- apprentissage permanent de nouveaux services → **source d'erreurs**

# Concept d'interface uniforme

Détails du protocole HTTP et de l'interface uniforme  $\Rightarrow$  voir les RFC 7230 à 7237

# Etat d'une application REST

« Etat d'une application » → état des ressources gérées par l'application et possibilités d'action sur ces ressources

**HATEOAS** : Hypertext As The Engine Of Application State



# Etat d'une application REST

En pratique : la réponse à une requête contient des **liens hypertexte** (URI) permettant de **lister** ou **manipuler** des ressources → navigation par URI.

L'état de l'application a changé après la requête, les nouvelles possibilités sont exprimées par ces nouvelles URI.

# Etat d'une application REST

Exemple de la réponse à un POST :

HTTP 1.1 201 OK

**Location:** /monblog/738025

L'application fournit au client une URI lui permettant de manipuler la ressource via cette URI (modification → PUT, suppression → DELETE)

# Etat d'une application REST

L'application peut aussi fournir un lien permettant de lister d'autres ressources associées à la ressource sur laquelle porte une requête GET

- ressources contenues par exemple (liste d'employés retournée par requête GET sur l'URI d'une société)

# Etat d'une application REST

## Exemple d'une réponse HTTP (en JSON) :

```
{  
  "departmentId": 10,  
  "departmentName": "Administration",  
  "locationId": 1700,  
  "managerId": 200,  
  . . .  
}
```

# Etat d'une application REST

```
"links": [ {  
    "href": "10/employees",  
    "rel": "employees",  
    "type" : "GET"  
} ]
```

```
}
```

# Etat d'une application REST

Le client recevant la réponse précédente peut suivre le lien indiqué par `href` (`10/employees`) et ainsi obtenir au travers d'une requête GET la ressource `employees` (liste des employés) associée à la ressource "département numéro 10 / Administration".

# Etat d'une application REST

Cette navigation au travers des liens fournis par le serveur est équivalente à ce que réalise un internaute en cliquant sur les liens fournis par les pages HTML d'un site.

# Pas d'état conversationnel côté serveur

Le serveur n'enregistre pas d'état conversationnel (infos de session) :

- pas d'association entre une activité côté serveur et une activité côté client
- pas d'informations sur l'état d'un client côté serveur
- une requête précise tous ses paramètres

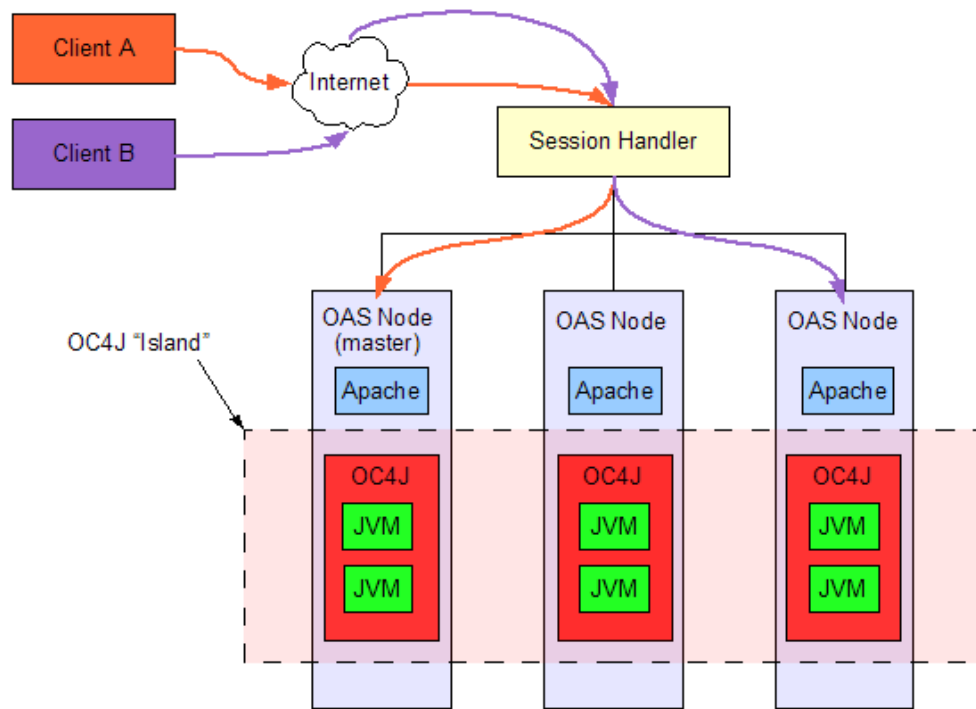


# Pas d'état conversationnel côté serveur

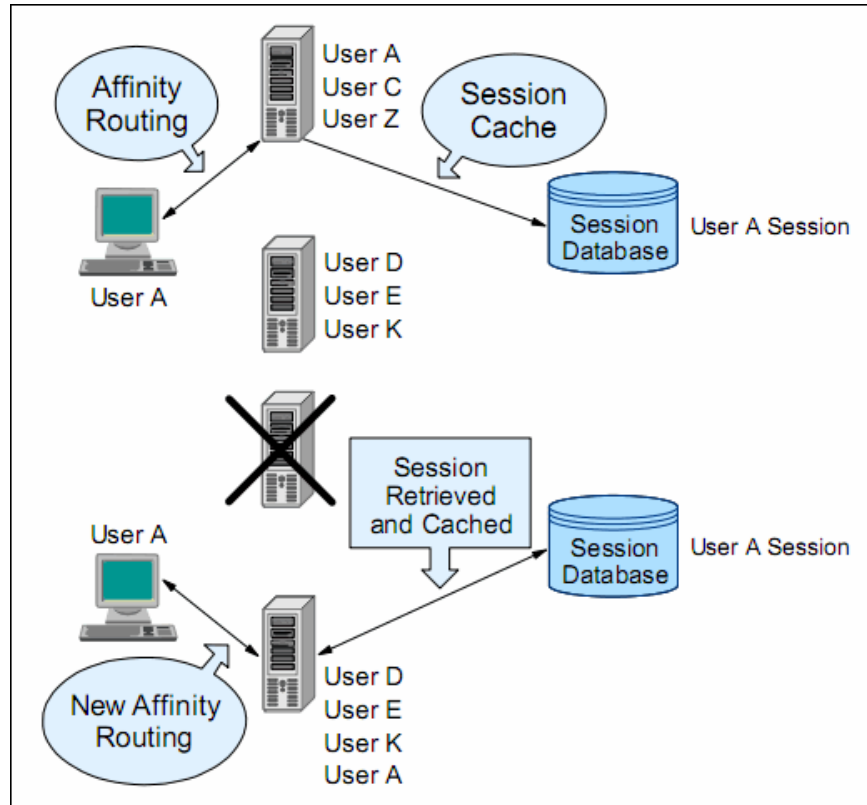
On dit qu'il n'y a pas d'affinité de session entre un **client** et une **activité côté serveur** (*session affinity* ou *sticky session* en anglais).

Le serveur doit donc fournir en retour au client toute les informations utiles pour qu'il gère cet état.

# Pas d'état conversationnel côté serveur



# Pas d'état conversationnel côté serveur



# Pas d'état conversationnel côté serveur

C'est au client de gérer l'état conversationnel (informations de session).

Il passe ces informations (si nécessaire) à chaque requête.

# Pas d'état conversationnel côté serveur

Intérêts de cette absence d'état :

- toute requête peut être traitée par n'importe quelle activité côté serveur ⇒ **scalabilité**, comme avec les EJB

Session stateless : des activités peuvent être démarrées ou arrêtées pour s'adapter à la demande

# Pas d'état conversationnel côté serveur

Intérêts de cette absence d'état :

- la défaillance d'une activité côté serveur a un impact limité au traitement en cours  
⇒ **robustesse** des applications

# Pas d'état conversationnel côté serveur

Intérêts de cette absence d'état :

- le serveur est plus simple puisqu'il n'a pas à gérer de sessions utilisateur ⇒  
**facilité de développement**

# Pas d'état conversationnel côté serveur

Intérêts de cette absence d'état :

- les requêtes sont lisibles car elles contiennent toute l'information nécessaire pour les comprendre ⇒ des **intermédiaires** peuvent les traiter (cache dans reverse proxies, par exemple)



# Pas d'état conversationnel côté serveur

Désavantages de cette absence d'état :

- le **trafic réseau** est augmenté (plus d'informations à passer à chaque requête)
- les clients sont **plus lourds** à développer

# Web services façon JAX-RS

JAX-RS est une API Java facilitant le développement d'applications REST

- à l'aide d'**annotations** Java

# Exemples d'annotations JAX-RS

## Annotations usuelles :

- `@Path` : path relatif permettant d'associer une **URI** à un traitement
- `@GET` / `@POST` / `@PUT` / ... : annotations sur méthode pour indiquer quelles requêtes HTTP peuvent être **traitées** par cette méthode

# Exemples d'annotations JAX-RS

## Annotations usuelles :

- `@PathParam` : accès à un **paramètre extrait** d'une URI
- `@FormParam` : accès à un **paramètre de formulaire** HTTP

# Exemples d'annotations JAX-RS

## Annotations usuelles :

- `@Context` : injection du **contexte général** d'une requête ou réponse HTTP
- `@Consumes` / `@Produces` : annotations sur méthode indiquant quel **type MIME** de donnée est produit ou consommé par cette méthode

# Exemple de code annoté

```
@Path("helloworld")
```

```
public class HelloWorld {  
    @Context  
    private UriInfo context;  
    public HelloWorld() {  
    }  
}
```

# Exemple de code annoté

**@GET**

**@Produces ("text/html")**

```
public String getHtml() {  
    return "<html  
lang=\"en\"><body><h1>Hello, World!!</h1></  
body></html>";  
}  
  
}
```

# Les annotations `@Path` et `@PathParam`

L'annotation `@Path` permet d'associer une URI à une ressource (classe) ou à une méthode.

Cette URI peut comporter des champs variables :

```
@Path("/users/{username}")
```



# Les annotations @Path et @PathParam

Un champ variable peut être récupéré au moyen de l'annotation @PathParam :

```
public String  
getUser (@PathParam ("username") String  
userName) {  
  
    ...  
  
}
```

# Les annotations @Path et @PathParam

```
@Path("/users/{username}")  
  
public class UserResource {  
    @GET  
  
    public String getUser(@PathParam("username")  
String username) {  
  
        ...  
  
    }  
  
}
```

# Les annotations @Path et @PathParam

```
@Path("/{name1}/{name2}/")  
public class SomeResource {  
    ...  
}
```

# L'annotation @QueryParam

L'annotation `@QueryParam` permet de récupérer un paramètre de requête HTTP :

`http://www.acme.com/temperature/limits?low=5&high=15`

# L'annotation @QueryParam

Il est possible de spécifier une valeur par défaut pour un paramètre de requête.

Si le paramètre n'est pas précisé dans la requête, la valeur indiquée dans `@PathParam` est utilisée.

# L'annotation @QueryParam

```
@Path("limits")
@GET
public Response setLimits(
    @DefaultValue("2") @QueryParam("low") int
    lowLimit,
    @DefaultValue("20") @QueryParam("high") int
    highLimit) {
    ...
}
```

# L'annotation `@RequestParam`

Un formulaire HTML peut transmettre des données dans une requête POST selon le type MIME `application/x-www-form-urlencoded`.

L'annotation `@RequestParam` permet de récupérer de tels paramètres.

# L'annotation `@FormParam`

```
@POST
```

```
@Consumes("application/x-www-form-  
urlencoded")
```

```
public void post(@FormParam("name")  
String name) {  
  
    ...  
  
}
```



# L'annotation @CookieParam

Les cookies sont un moyen de transmettre des informations entre un client et un serveur HTTP.

Ils peuvent être obtenus dans un web service REST au moyen de l'annotation @CookieParam.

# L'injection de contexte HTTP

Plus généralement, il est possible d'accéder par injection au contexte d'un échange HTTP :

- paramètres de requête et de formulaire
- cookies
- segments d'URI
- headers HTTP
- ...

# L'injection de contexte HTTP

Ceci est réalisé en utilisant l'annotation `@Context` et en indiquant le type de l'information recherchée :

- `UriInfo`
- `HttpHeaders`
- `SecurityContext`
- `Providers`

# L'injection de contexte HTTP

@GET

```
public String get(@Context HttpHeaders  
hh) {  
    MultivaluedMap<String, String>  
headerParams = hh.getRequestHeaders();  
  
    Map<String, Cookie> pathParams =  
hh.getCookies();  
}
```

# Réaliser un web service REST

Un web service REST doit contenir au moins une classe-ressource packagée dans un fichier WAR.

L'URI de base à laquelle le web service répond doit être définie :

- par annotation `@ApplicationPath`
- ou par configuration du fichier `web.xml`

# Réaliser un web service REST

Une classe-ressource annotée par  
`@ApplicationPath` doit dériver de  
`javax.ws.rs.core.Application`

# Réaliser un web service REST

```
package com.acme;  
  
@ApplicationPath("/banque")  
  
public class Banque extends  
Application { ... }
```

# Réaliser un web service REST

Les URI activant le service web précédent sont relatives à /banque :

`/banque/compte06749`

`/banque/clients`

`/banque/transfert/5643653`



# Réaliser un web service REST

La définition suivante définit les URI traitées par la classe `com.acme.Banque` :

```
<servlet-mapping>  
    <servlet-name>com.acme.Banque</servlet-name>  
    <url-pattern>/banque/*</url-pattern>  
</servlet-mapping>
```

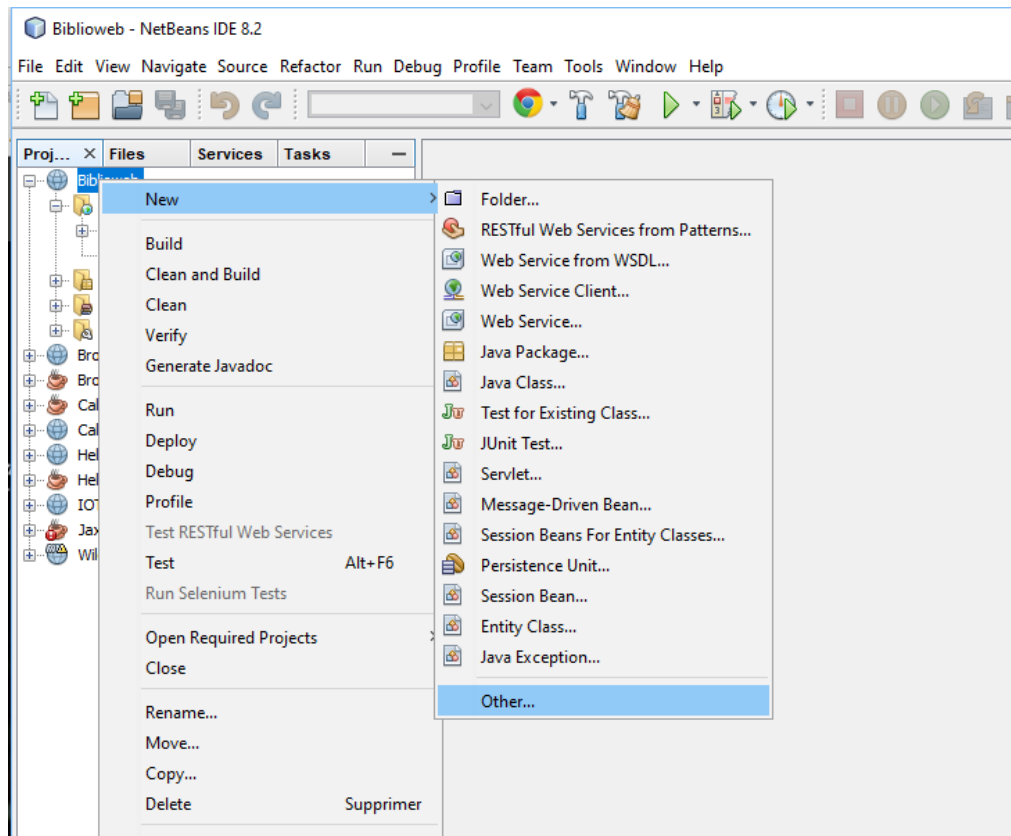
# Réaliser un web service REST

Cette annotation **surcharge** l'annotation `@ApplicationPath`.

Il est ainsi possible de redéfinir les URI traitées par un web service sans modifier le code source.

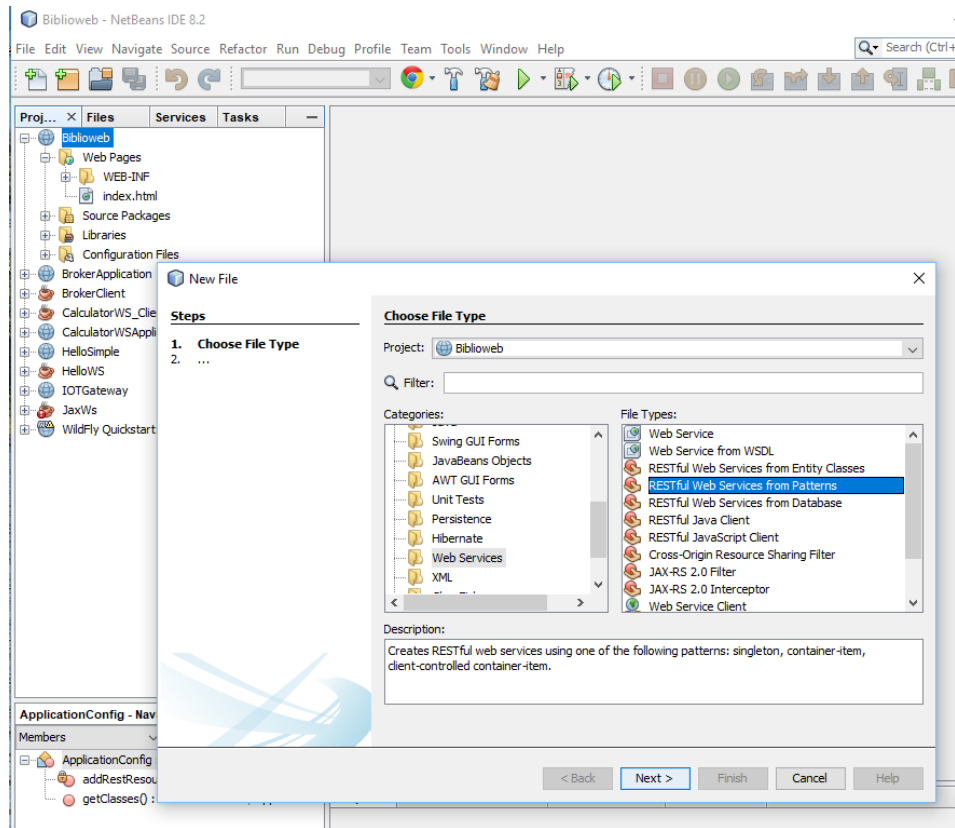
# Développement d'un web service REST

## Création du web service (étape 1)



# Développement d'un web service REST

## Création du web service (étape 2)



# Développement d'un web service REST

## Création du web service (étape 3)

New RESTful Web Services from Patterns

**Steps**

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

**Select Pattern**

Select a RESTful web service design pattern:

☒ Simple Root Resource

☐ Container-Item

☐ Client-Controlled Container-Item

**Description:**

Create a RESTful root resource class with GET and PUT methods using Java API for RESTful Web Service (JSR-311). This pattern is useful for creating a simple HelloWorld service and wrapper services for invoking WSDL-based web services.

On the next page you will be specifying class name, URI, and representation type of the resource.

< Back   Next >   Finish   Cancel   Help

# Développement d'un web service REST

## Création du web service (étape 4)

New RESTful Web Services from Patterns

**Steps**

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

**Specify Resource Classes**

Project: Biblioweb

Location: Source Packages

Resource Package: biblio.web

Path: generic

Class Name: GenericResource

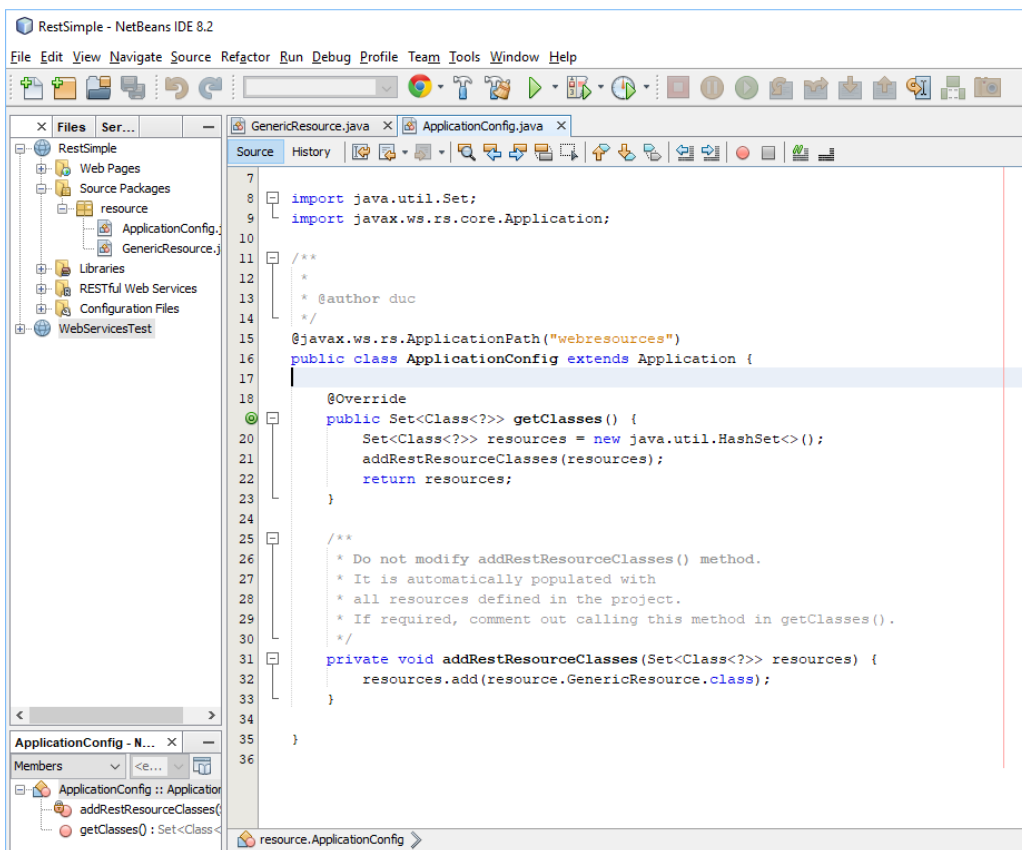
MIME Type: application/xml

Representation Class: java.lang.String Select...

< Back Next > **Finish** Cancel Help

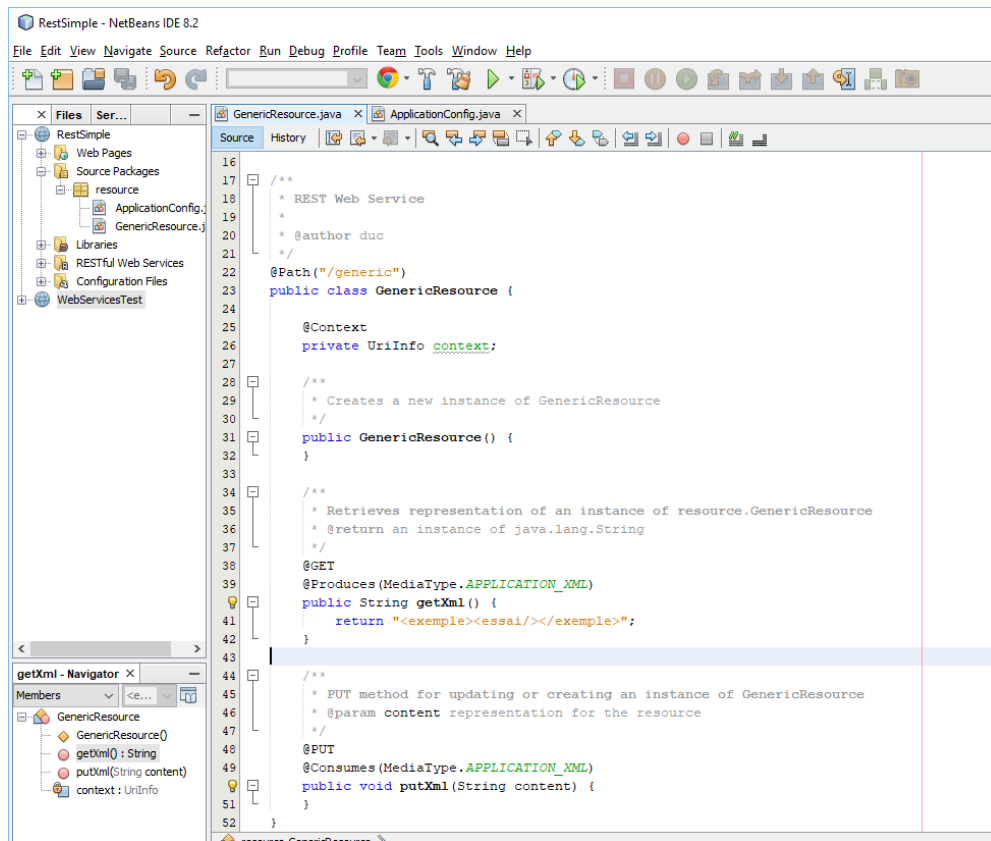
# Développement d'un web service REST

Création du web service (classe Application produite)



# Développement d'un web service REST

Création du web service (classe ressource produite)



```
16
17 /**
18  * REST Web Service
19  *
20  * @author duc
21  */
22 @Path("/generic")
23 public class GenericResource {
24
25     @Context
26     private UriInfo context;
27
28     /**
29      * Creates a new instance of GenericResource
30      */
31     public GenericResource() {
32     }
33
34     /**
35      * Retrieves representation of an instance of resource.GenericResource
36      * @return an instance of java.lang.String
37      */
38     @GET
39     @Produces(MediaType.APPLICATION_XML)
40     public String getXml() {
41         return "<example><essai/></example>";
42     }
43
44     /**
45      * PUT method for updating or creating an instance of GenericResource
46      * @param content representation of the resource
47      */
48     @PUT
49     @Consumes(MediaType.APPLICATION_XML)
50     public void putXml(String content) {
51     }
52 }
```

RestSimple - NetBeans IDE 8.2

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

RestSimple

- Web Pages
- Source Packages
  - resource
    - ApplicationConfig.java
    - GenericResource.java
- Libraries
- RESTful Web Services
- Configuration Files
- WebServicesTest

getXml - Navigator

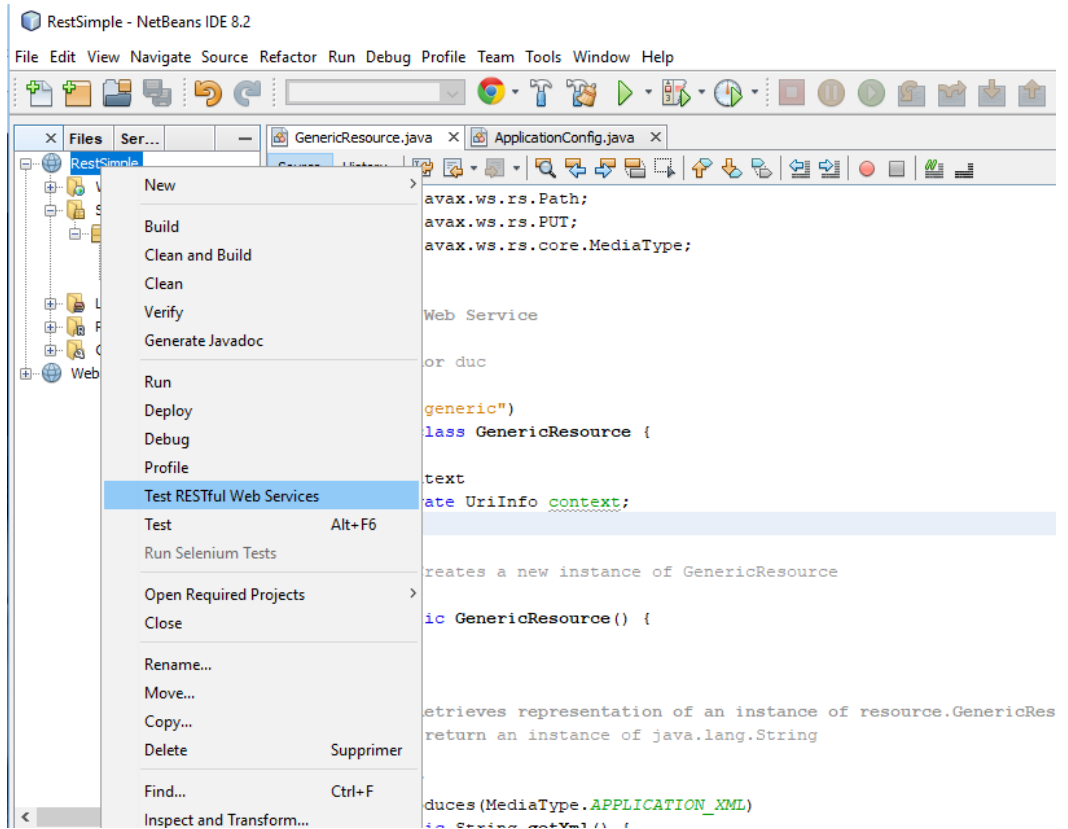
Members

- GenericResource
  - GenericResource()
  - getXml(): String
  - putXml(String content)
  - context: UriInfo



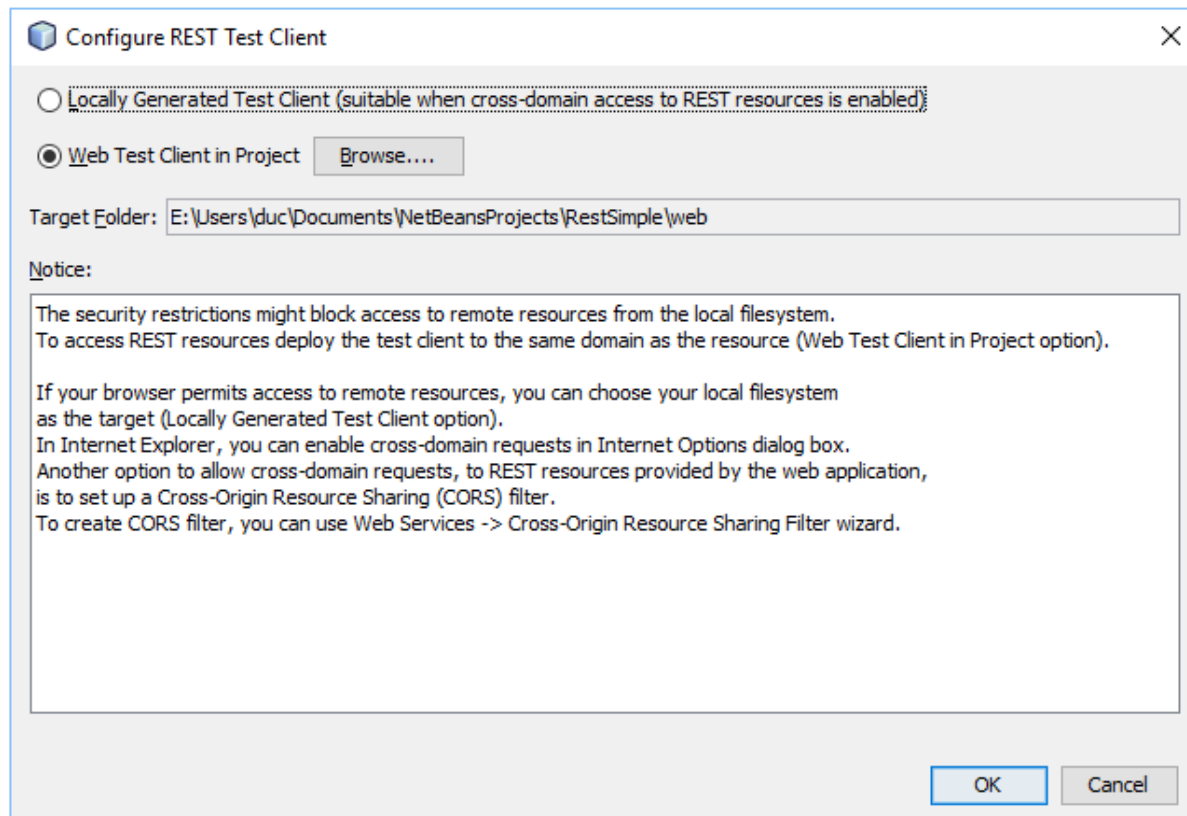
# Test d'un web service REST

## Activation du test (étape 1)



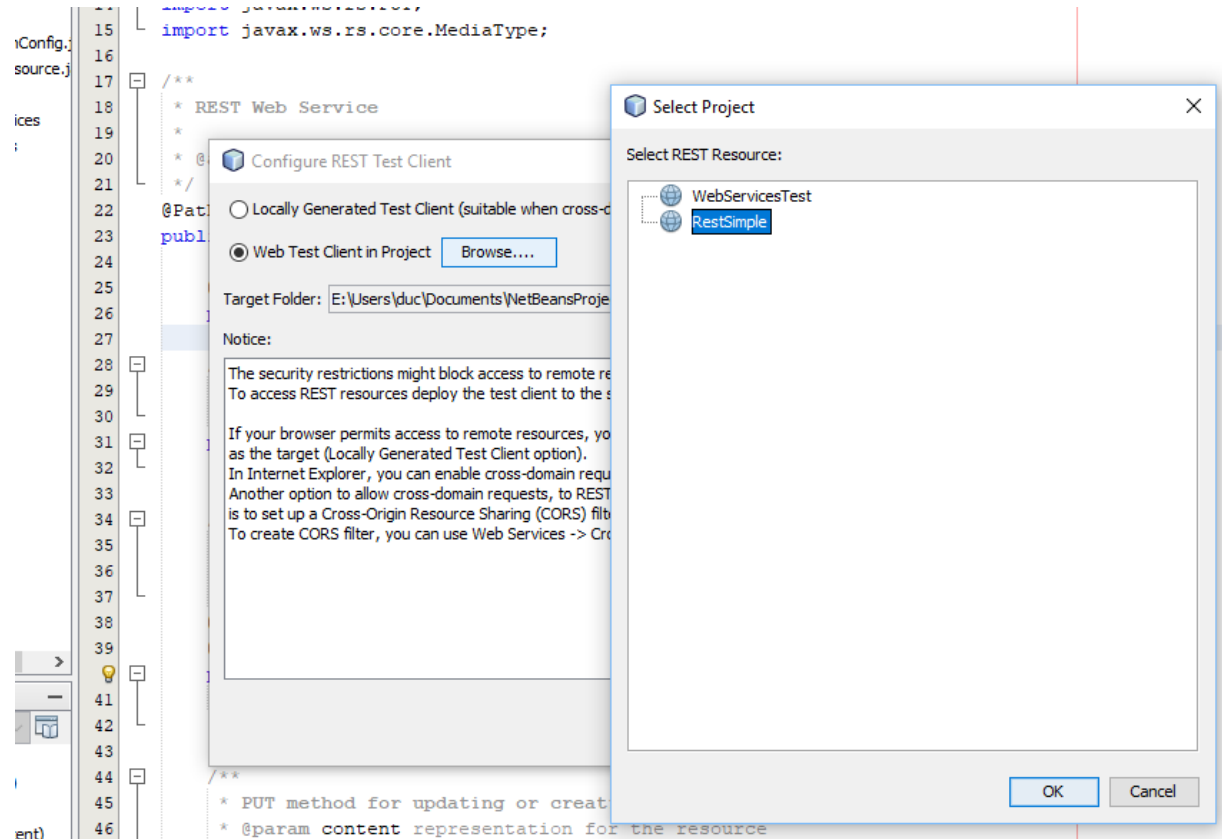
# Test d'un web service REST

## Activation du test (étape 2)



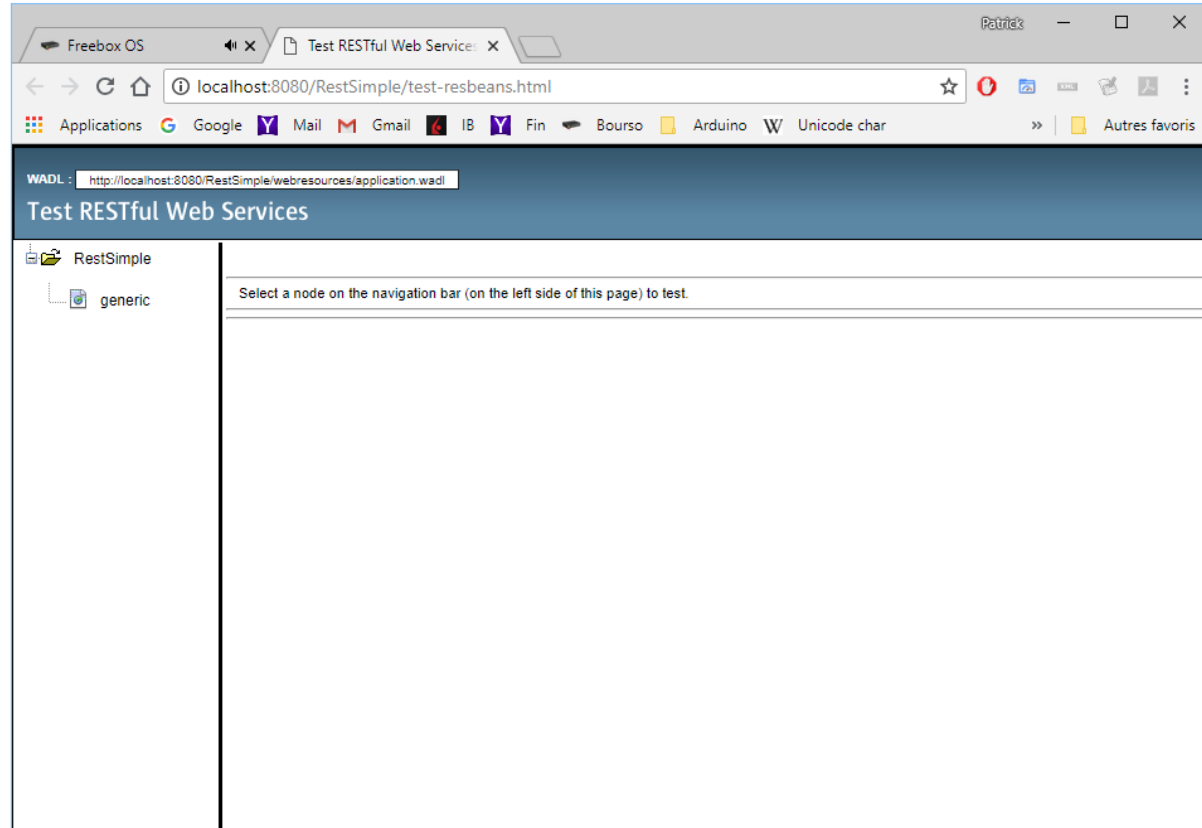
# Test d'un web service REST

## Activation du test (étape 3)



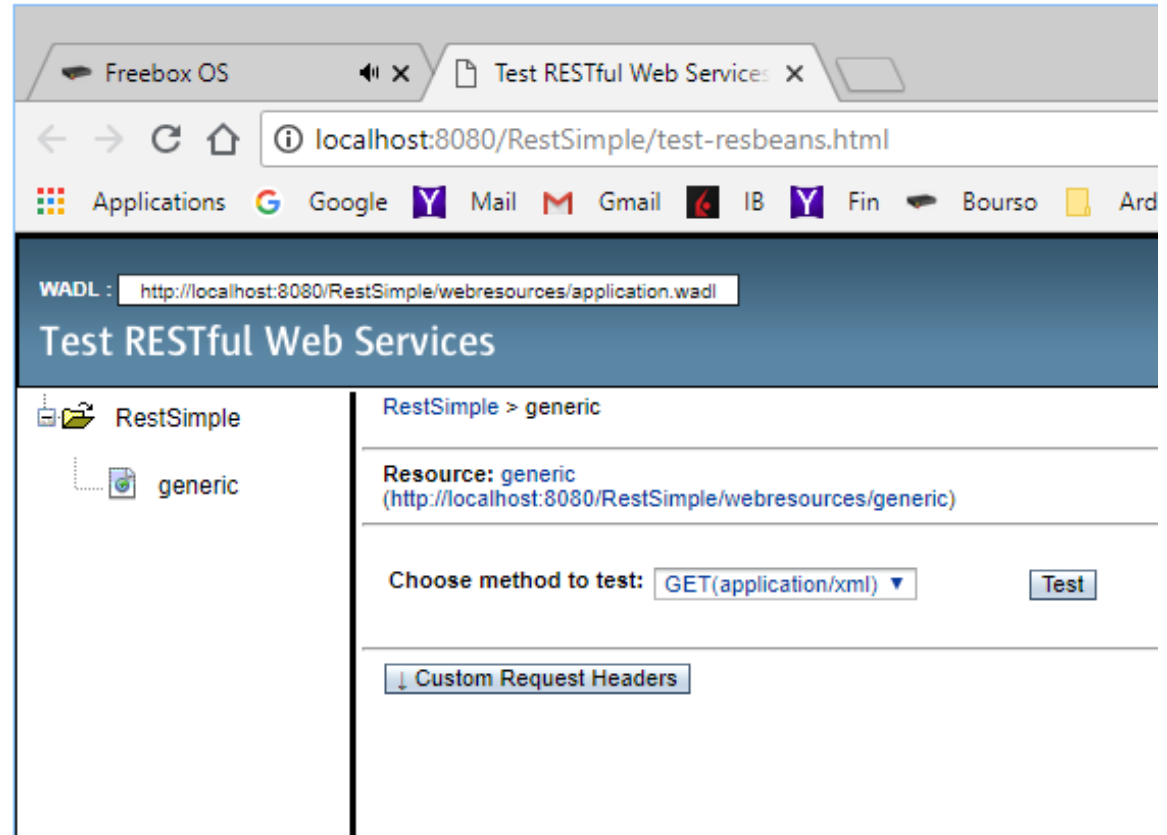
# Test d'un web service REST

## Test du web service REST



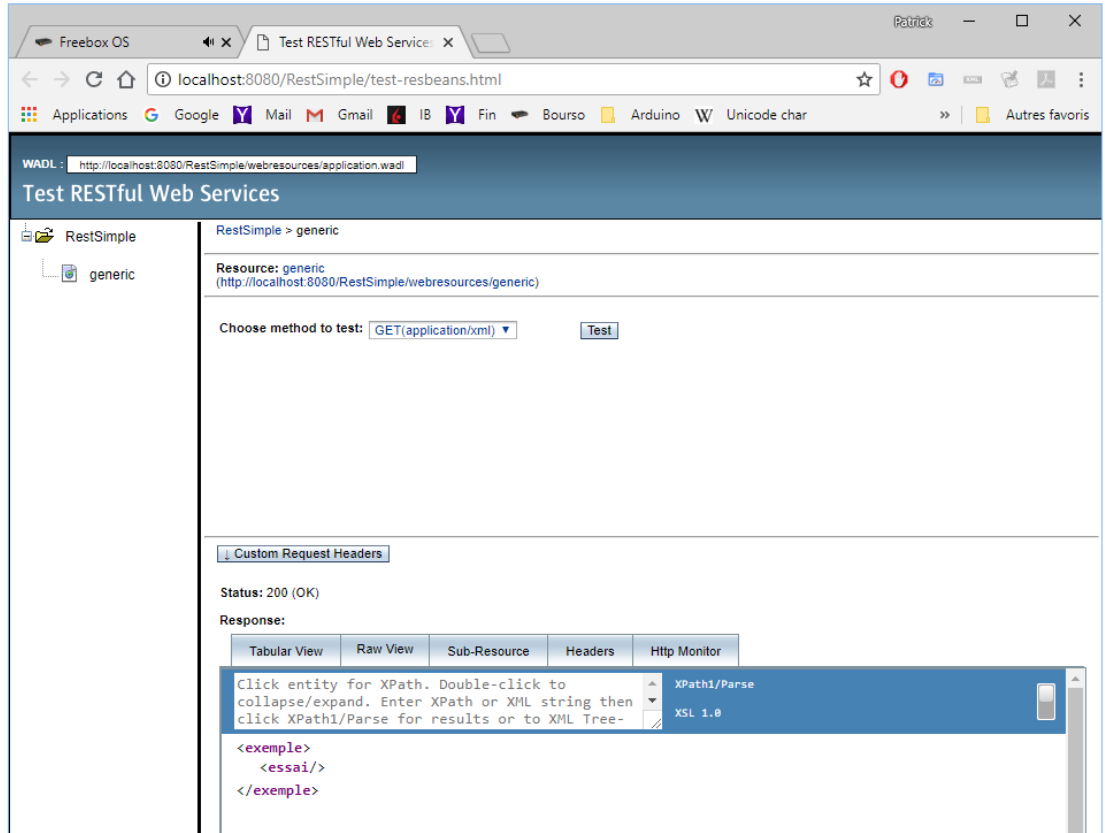
# Test d'un web service REST

## Test du web service REST



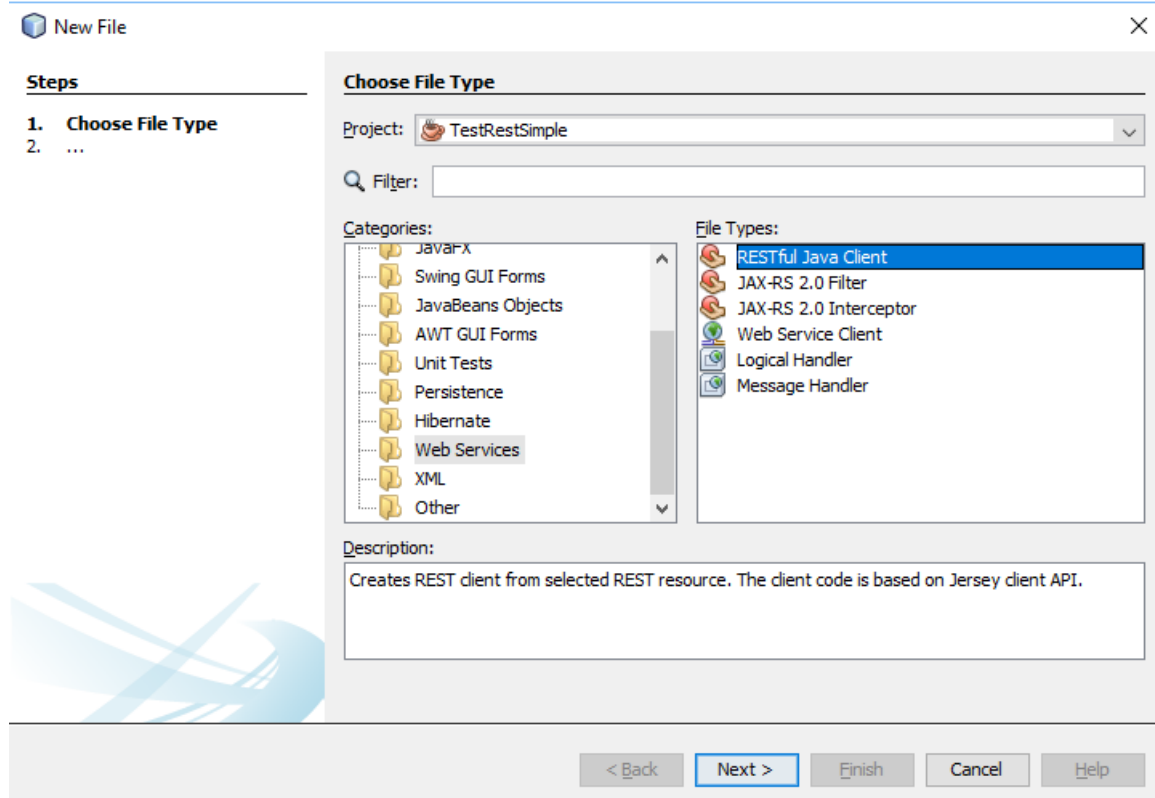
# Test d'un web service REST

## Test du web service REST



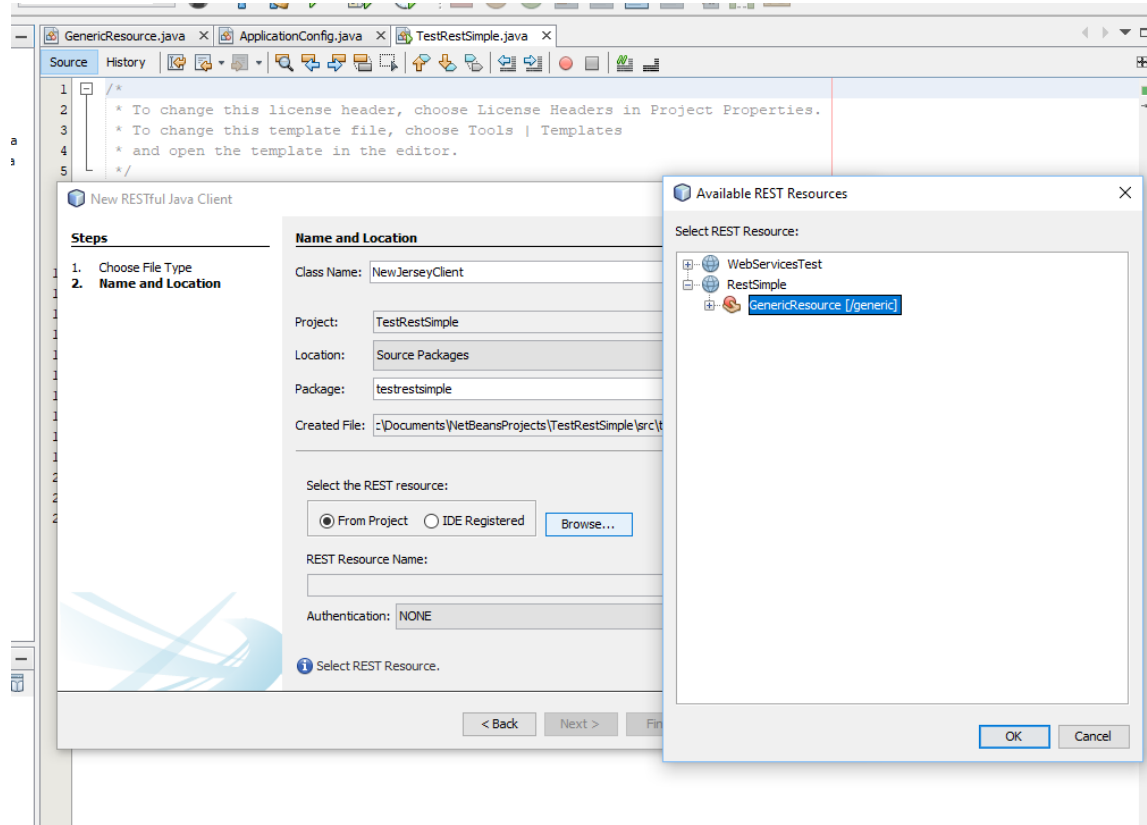
# Développement d'un client REST

## Développement d'un client de web service REST (étape 1)



# Développement d'un client REST

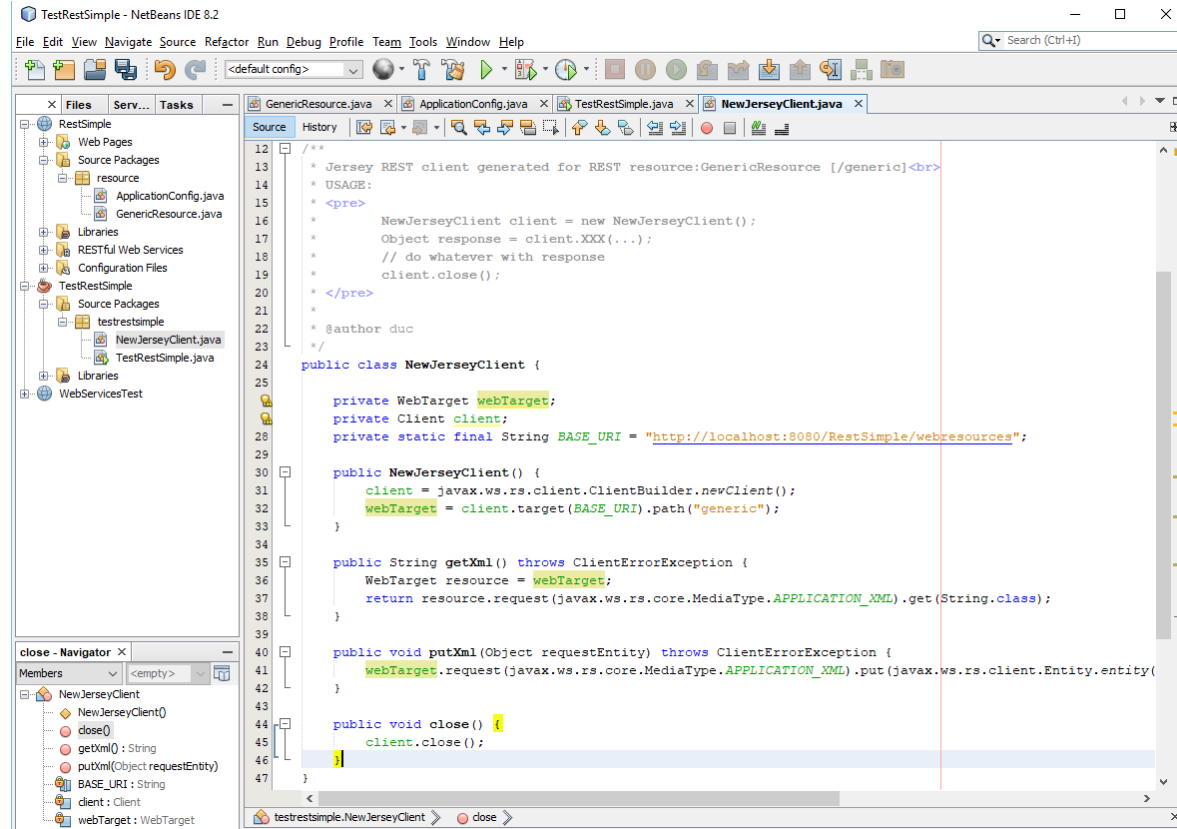
## Développement d'un client de web service REST (étape 2)





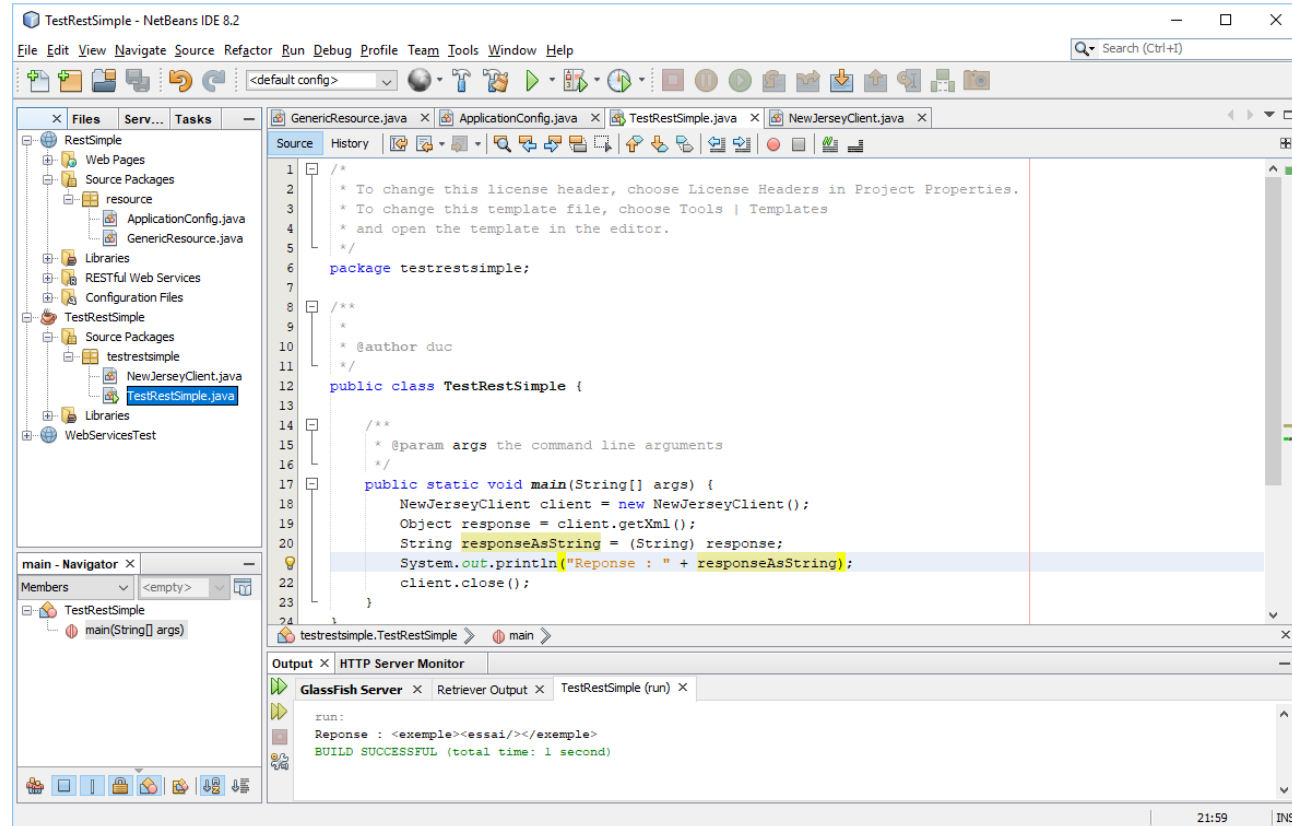
# Développement d'un client REST

## Développement d'un client de web service REST (étape 3)



# Développement d'un client REST

## Développement d'un client de web service REST (étape 4)



# Sémantique des verbes HTTP

GET : récupération d'une représentation d'une ressource

- Paramètre : URI de la ressource
- Le type MIME de la représentation à retourner peut être spécifié par la directive HTTP `Accept`

# Sémantique des verbes HTTP

GET : récupération d'une représentation d'une ressource

- Opération idempotent et pas de changement côté serveur
- Code de retour normal : 200 OK, et réponse contenant la représentation de la ressource

# Sémantique des verbes HTTP

POST : création ou modification d'une ressource

- Paramètre : la (nouvelle) représentation de la ressource à créer ou à modifier
- Le type MIME de la représentation peut être spécifié par la directive HTTP `Content-Type`

# Sémantique des verbes HTTP

POST : création ou modification d'une ressource

- Opération non idempotent, modification côté serveur
- Code de retour normal : 201 Created (avec champ `Location`), 200 OK ou 204 No Content

# Sémantique des verbes HTTP

PUT : création ou modification d'une ressource nommée par le client

- Paramètres : la (nouvelle) représentation de la ressource à créer ou à modifier, et l'URI de la ressource (header `Request-URI`)
- Le type MIME de la représentation peut être spécifié par la directive HTTP `Content-Type`

# Sémantique des verbes HTTP

PUT : création ou modification d'une ressource nommée par le client

- Opération idempotent, modification côté serveur
- Code de retour normal : 201 Created, 200 OK ou 204 No Content



# Sémantique des verbes HTTP

DELETE : suppression d'une ressource

- Paramètre : l'URI de la ressource (header `Request-URI`)
- Codes de retour normal : 200 OK si suppression réalisée et status fourni dans la réponse ou 202 Accepted si la suppression aura lieu plus tard ou 204 No Content si suppression réalisée et pas de status fourni dans la réponse

# Sémantique des réponses HTTP

## 1xx : information

- Réponse contient un header Status-Line
- 100 Continue
- 101 Switching Protocols

# Sémantique des réponses HTTP

## 2xx : opération réussie

- 200 OK
- 201 Created
- 202 Accepted
- 203 Non-Authoritative Information
- 204 No Content

# Sémantique des réponses HTTP

## 3xx : redirection

- 300 Multiple Choices
- 301 Moved Permanently
- 302 Found
- 303 See Other
- 304 Not Modified

# Sémantique des réponses HTTP

## 4xx : erreur du client

- 400 Bad Request
- 401 Unauthorized
- 402 Payment Required
- 403 Forbidden
- 404 Not Found

# Sémantique des réponses HTTP

## 5xx : erreur du serveur

- 500 Internal Server Error
- 501 Not Implemented
- 502 Bad Gateway
- 503 Service Unavailable
- 504 Gateway Timeout

# Génération et analyse de réponse HTTP

JAX-RS fournit trois classes-utilitaires :

- `javax.ws.rs.core.Response`

==> permet de créer une réponse HTTP (côté serveur) ou d'analyser une réponse HTTP (côté client)

# Génération de réponse par le serveur

- `javax.ws.rs.core.Response.ResponseBuilder`

**==> permet de construire une réponse HTTP contenant des métadonnées et / ou un body**



# Génération de réponse par le serveur

- `javax.ws.rs.core.UriBuilder`  
==> permet de construire une URI

# Exemple de génération de réponse HTTP

**@POST**

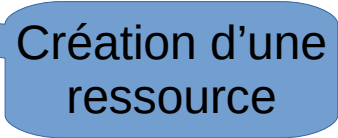
Exemple d'une  
opération POST

```
Response addWidget(...) {  
    Widget w = ...  
  
    URI widgetId =  
    UriBuilder.fromResource(Widget.class) ...  
    return Response.created(widgetId).build();  
}
```

# Exemple de génération de réponse HTTP

@POST

```
Response addWidget(...) {  
    Widget w = ...  
    URI widgetId =  
    UriBuilder.fromResource(Widget.class) ...  
    return Response.created(widgetId).build();  
}
```




Création d'une ressource

# Exemple de génération de réponse HTTP

@POST

```
Response addWidget(...) {  
    Widget w = ...  
    URI widgetId =  
UriBuilder.fromResource(Widget.class) ...  
    return Response.created(widgetId).build();  
}
```

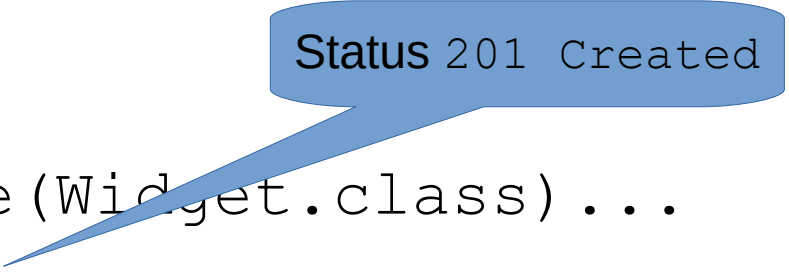


URI de la  
ressource créée

# Exemple de génération de réponse HTTP

@POST

```
Response addWidget(...) {  
    Widget w = ...  
    URI widgetId =  
    UriBuilder.fromResource(Widget.class) ...  
    return Response.created(widgetId).build();  
}
```

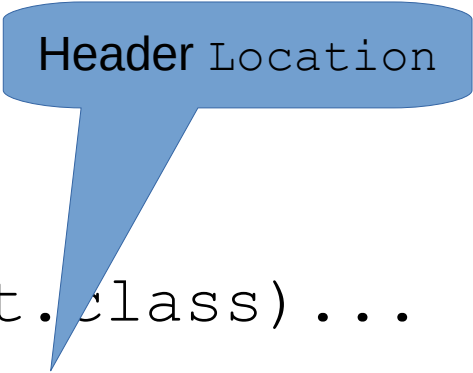


Status 201 Created

# Exemple de génération de réponse HTTP

@POST

```
Response addWidget(...) {  
    Widget w = ...  
    URI widgetId =  
    UriBuilder.fromResource(Widget.class) ...  
    return Response.created(widgetId).build();  
}
```



# Exemple de génération de réponse HTTP

@POST

```
Response addWidget(...) {  
    Widget w = ...  
    URI widgetId =  
    UriBuilder.fromResource(Widget.class)..  
    return Response.created(widgetId) .build() ;  
}
```

Construction de  
l'objet réponse HTTP

# Pour en savoir plus sur HTTP

RFC 2616 :

<https://tools.ietf.org/html/rfc2616>



# Documentation de web service REST

**Title :** Show All Users.

**URL :** /users **or** /users/{id} **or** /users?id={id}

**Method :** GET | POST | DELETE | PUT

**URL Params :** Required: id=[integer] **OR** Optional:  
photo\_id=[alphanumeric]

**Data Params :** { u : { email : [string], name : [string],  
current\_password : [alphanumeric] password :  
[alphanumeric], password\_confirmation : [alphanumeric] } }

**Response Codes:** Success (200 OK), Bad Request (400), Unauthorized (401)

# Web services façon JAX-RS : exercice

Réaliser l'exercice 15 (bibliothèque électronique)

# Pour approfondir

La gestion des logs applicatifs

# Information sur l'état d'une application

Nécessité de savoir ce qui se passe dans une application logicielle

- généralement impossible d'utiliser un debugger
- plusieurs applications peuvent **concourir** pour réaliser une fonction

# Information sur l'état d'une application

Différents contextes d'utilisation d'une application  $\Rightarrow$  différentes populations :

- développement
- intégration
- validation
- production

# Information sur l'état d'une application

Accès aux informations plus ou moins compliqué selon le type d'application et son contexte d'utilisation :

- application serveur
- application hébergée
- application avec IHM
- ...

# Information sur l'état d'une application

Seule solution en général : l'enregistrement de logs applicatifs dans un fichier d'évènements (fichier de log).

**Evènement** → génération d'un ou plusieurs **logs applicatifs** dans un **fichier de logs** (fichier d'évènements)

# La gestion des logs applicatifs

Evènement système ou réseau :

- démarrage / arrêt de la machine
- login / logout d'un utilisateur
- dysfonctionnement d'un équipement matériel
- etc.



# La gestion des logs applicatifs

## Evènement applicatif :

- démarrage / arrêt de l'application
- début / fin de traitement fonctionnel
- production d'un résultat
- erreur de traitement, manque de mémoire
- etc.

# La gestion des logs applicatifs

Ce mécanisme est utilisé par tous les composants logiciels bien faits : modules système, modules réseau, utilitaires divers, outils d'utilisateur final, etc.

# Informations enregistrées

Un log applicatif doit idéalement comporter certaines informations “standard” :

- date / heure d’occurrence (précision en deça de la seconde)
- numéro de processus et thread concerné
- **sévérité** de l’évènement

# Informations enregistrées

- message explicatif de l'évènement
- nom du fichier source et numéro de la ligne produisant le log dans certains contextes

# La gestion des logs applicatifs

Il est souvent nécessaire d'associer les évènements de diverses applications ou outils ⇒ **agrégation de logs** par date / heure

# Types d'évènement

Tout évènement notable doit être enregistré :

- normal
- inattendu
- dysfonctionnement
- etc.

# Sévérité d'un évènement

## Sévérités habituelles :

- critique (**critical**)  $\Rightarrow$  l'application doit s'arrêter
- erreur (**error**)  $\Rightarrow$  une fonction ne peut être réalisée mais l'application peut continuer à fonctionner

# Sévérité d'un évènement

## Sévérités habituelles :

- avertissement (**warning**)  $\Rightarrow$  une fonction peut être remplie, mais quelque chose d'inattendu se produit
- information (**information**)  $\Rightarrow$  évènement normal



# Sévérité d'un évènement

## Sévérités habituelles :

- debugging (**debug**)  $\Rightarrow$  information détaillée sur une étape de traitement
- très détaillé (**verbose**)  $\Rightarrow$  détail poussé sur une étape de traitement

# Populations utilisant les logs

Utilisateurs finaux (utilisent l'application pour ses fonctions)

- intéressés par les évènements fonctionnels normaux, les erreurs fonctionnelles et les erreurs critiques

# Populations utilisant les logs

Opérateurs (gèrent l'infrastructure sur laquelle s'exécute l'application)

- principalement intéressés par évènements nominaux et les erreurs critiques

# Populations utilisant les logs

Intégrateurs/valideurs (intègrent l'application avec d'autres applications ou sur une infrastructure de test et la valident)

- intéressés par tous les évènements

# Populations utilisant les logs

Développeurs (développent et testent l'application, assurent le support pour les intégrateurs/valideurs, maintiennent l'application en production)

- intéressés par tous les événements

# Politique de logging

Il est très utile de diriger les logs vers au moins deux sorties indépendantes :

- l'une pour les logs destinés aux utilisateurs finaux, opérateurs et intégrateurs/valideurs
- l'autre pour les développeurs et intégrateurs/valideurs

# Politique de logging

Très important : les messages explicatifs de niveau INFO et supérieurs doivent être significatifs pour des utilisateurs finaux et opérateurs

- pas de message du genre

```
MaClass : invalid pointer
```

```
exit loop, not found
```

```
...
```

# Politique de logging

## Par contre :

```
export files for destinee « xxx »  
sucessfully produced
```

```
could not cipher data, reason is :  
could not connect to HSM (exception :  
coud not resolve hostname « host-1 »)
```



# Outils de logging

Dans le monde Java :

- log4j
- logback
- java.util.Logging
- ...

# Outils de logging

## Recommandé :

- slf4j
  - façade vers implémentations d'outils comme log4j, logback ou `java.util.Logging`
- log4j version 2 : également façade vers d'autres implémentations

# Outils de logging

Les performances d'une application peuvent être **plombées** par un outil de logging inefficace ou des messages de logging trop nombreux / volumineux



# Outils de logging

Il est important que l'outil de logging permette de diriger les logs vers différentes destinations, en se basant sur le niveau de logging ou autre filtrage.

# Outils de logging

## Exemple :

- sortie vers l'infra de logging système  
(syslog sur Linux, Event Logging API sur Windows)
- sortie vers fichier normal
- sortie vers socket réseau
- ...

# MERCI !

& SUIVEZ-NOUS !



HUMAN **booster**  
•• VOTRE SOLUTION COMPÉTENCE

04 73 24 93 11

[contact@humanbooster.com](mailto:contact@humanbooster.com)

[www.humanbooster.com](http://www.humanbooster.com)