REST (Representational State Transfert) est un type d'architecture reposant sur le fonctionnement même du web, qu'il applique aux services web. Pour concevoir un **service REST**, il faut bien connaître tout simplement le protocole **HTTP** (Hypertext Transfert Protocole), le principe des **URI** (Uniform Resource Identifiers) et respecter quelques règles. Il faut raisonner en terme de ressources.

PRÉSENTATION DES SERVICES WEB REST

Dans l'architecture REST, toute information est une ressource et chacune d'elles est désignée par une **URI** - généralement un lien sur le Web. Les ressources sont manipulées par un ensemble d'opérations simples et bien définies. L'architecture client-serveur de **REST** est conçue pour utiliser un protocole de communication sans état - le plus souvent **HTTP**. Ces principes encouragent la simplicité, la légèreté et l'efficacité des applications.

- **Les ressources** jouent un rôle central dans les architectures **REST**. Une ressource est tout ce que peut désigner ou manipuler un client, tout information pouvant être référencée dans un lien hypertexte. Elle peut être stockée dans une base de données, un fichier, etc. Il faut éviter autant que possible d'exposer des concepts abstraits sous forme de ressources et privilégier plutôt les objets simples.
- **URI** : Une ressource web est identifiée par une **URI**, qui est un identifiant unique formé d'un nom et d'une adresse indiquant où trouver la ressource. Il existe différents types d'URI : les adresses web, les **UDI** (Universal Document Identifiers), les **URI** (Uniform Resource Identifiers) et, enfin, les combinaisons d'**URL** (Uniform Resource Locator) et d'**URN** (Uniform Resource Name). Voici quelques exemples d'**URI** :
  - http://www.movies.fr/categories/aventure
  - http://www.movies.fr/catalog/titles/movies/123456
  - http://www.weather.com/weather/2012?location=Aurillac,France
  - http://www.flickr.com/explore/intersting/2012/01/01
  - http://www.flickr.com/explore/intersting/24hours

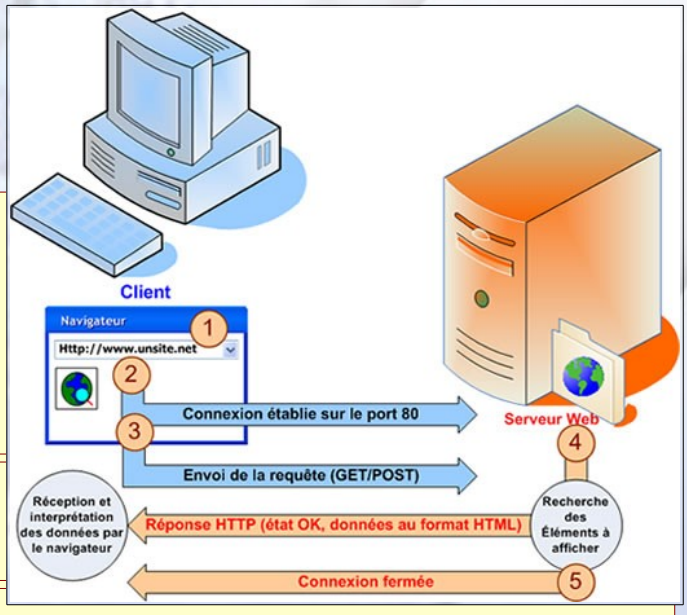
Les **URI** devraient être aussi descriptives que possible et ne désigner qu'une seule ressource, bien que des **URI** différentes qui identifient des ressources différentes puissent mener aux mêmes données : à un instant donné, la liste des photos intéressantes publiées sur Flickr le 01/01/2012 était la même que la liste des photos données au cours des 24 dernières heures, bien que l'information envoyée par les deux **URI** ne fût pas identique.

- **Représentation** : Nous pouvons obtenir la représentation d'un objet sous forme de texte, de **XML** de **PDF** ou tout autre format. Un client traite toujours une ressource au travers de sa représentation ; la ressource elle-même reste sur le serveur. La représentation contient toutes les informations utiles à propos de l'état d'une ressource :
  - http://www.unsite.fr/livres/catalogue/java
  - http://www.unsite.fr/livres/catalogue/java.csv
  - http://www.unsite.fr/livres/catalogue/java.xml

La première URI est la représentation par défaut de la ressource, les représentations supplémentaires lui ajoutent simplement l'extension de leur format : **.csv, .xml, .pdf, etc.**

PROTOCOLE HTTP

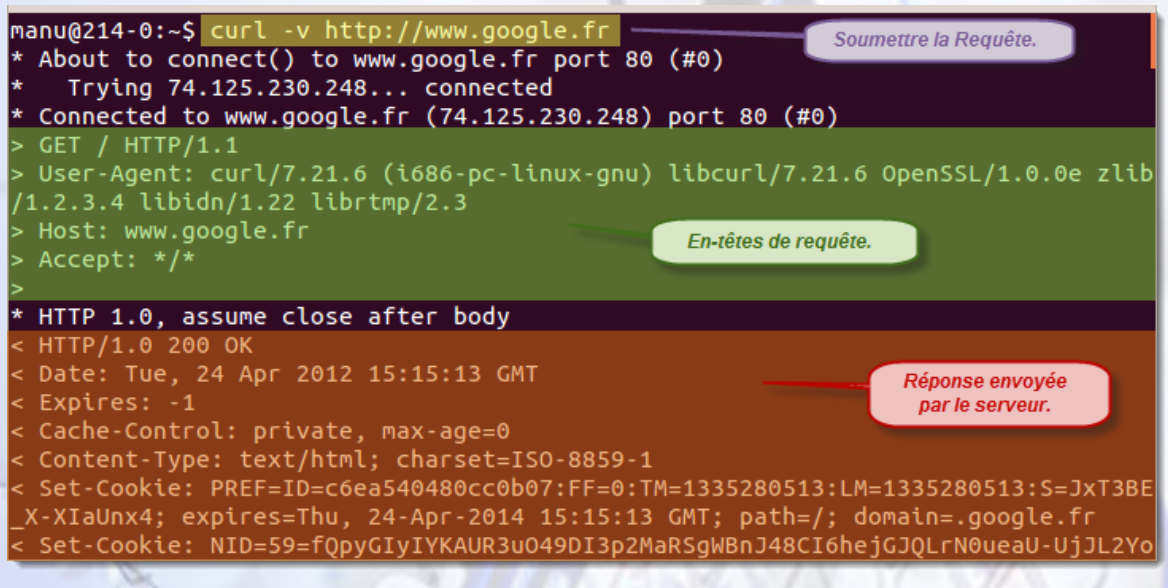
La requête la plus simple du protocole **HTTP** est formé de **GET** suivi d'une **URL** qui pointe sur des données (fichier statiques, traitement dynamique...). Elle est envoyée par un navigateur quand nous saisissons directement une URL dans le champ d'adresse du navigateur. Le serveur **HTTP** répond en renvoyant les données demandées.



- En tapant l'URL d'un site, l'internaute envoie (via le navigateur) une requête au serveur.
- Une connexion s'établit entre le client et le serveur sur le port 80 (port par défaut d'un serveur Web).
- Le navigateur envoie une requête demandant l'affichage d'un document. La requête contient entre autres la méthode (**GET, POST, etc.**) qui précise comment l'information est envoyée.
- Le serveur répond à la requête en envoyant une réponse **HTTP** composée de plusieurs parties, dont :
  - l'état de la réponse, à savoir une ligne de texte qui décrit le résultat du serveur (code ***200*** pour un accord, ***400*** pour une erreur due au client, ***500*** pour un erreur due au serveur) ;
  - les données à afficher.
- Une fois la réponse reçue par le client, la connexion est fermée. Pour afficher une nouvelle page du site, une nouvelle connexion doit être établie.

✗ REQUÊTES ET RÉPONSES

Un client envoie une requête à un serveur afin d'obtenir une réponse. Les messages utilisés pour ces échanges sont formés d'une enveloppe et d'un corps également appelé document ou représentation. Voici, par exemple, un type de requête envoyée à un serveur :



- ✗ La méthode **HTTP GET** ;
- ✗ Le chemin, ici la racine **"/"** ;
- ✗ Plusieurs autre en-têtes de requête.

*Vous remarquez que la requête n'a pas de corps (un **GET** n'a jamais de corps). En réponse, le serveur renvoie sa réponse et elle est formée des parties suivantes :*

- ✗ Un code de réponse : ici le code est **200 OK**.
- ✗ Plusieurs en-têtes de réponse, notamment **Date**, **Server**, **Content-Type**. Ici, le type de contenu est ***text/html***, mais il pourrait s'agir de n'importe quel format comme du **XML** (***application/xml***) ou une **image** (***image/jpeg***), etc.
- ✗ Un corps ou représentation. Ici, il s'agit du contenu de la page web renvoyée (qui n'est pas visible sur la figure proposée ci-dessus).

✗ RAPPEL SUR LES MÉTHODES DU HTTP

Le web est formé de ressources bien identifiées, reliées ensemble et auxquelles accéder au moyen de requêtes HTTP simples. Les requêtes principales de **HTTP** sont de type **GET, POST, PUT et DELETE**. Ces types sont appelés verbes ou méthodes. **HTTP** définit quatre autres verbes plus rarement utilisés, **HEAD, TRACE, OPTIONS** et **CONNECT**.

- ✗ ***GET*** : est une méthode de lecture demandant une représentation d'une ressource. Elle doit être implémentée de sorte à ne pas modifier l'état de la ressource. En outre, **GET** doit être idempotente, ce qui signifie qu'elle doit laisser la ressource dans le même état, quel que soit le nombre de fois où elle est appelée. Ces deux caractéristiques garantissent une plus grande stabilité : si un client n'obtient pas de réponse (à cause d'un problème réseau, par exemple), il peut renouveler sa requête et s'attendre à la même réponse que celle qu'il aurait obtenue initialement, sans corrompre l'état de la ressource sur le serveur.
- ✗ ***POST*** : A partir d'une représentation texte, XML, etc., **POST** crée une nouvelle ressource subordonnée à une ressource principale identifiée par l'**URI** demandée. Des exemples d'utilisation de **POST** sont l'ajout d'un message à un fichier journal, d'un livre à une liste d'ouvrages, etc. **POST** modifie donc l'état de la ressource et n'est pas idempotente (envoyer deux fois la même requête produit deux nouvelles ressources subordonnées). Si une ressource a été créée sur le serveur d'origine, le code de la réponse devrait être **201** (***Created***). La plupart des navigateurs modernes ne produisent que des requêtes **GET** et **POST**.
- ✗ ***PUT*** : Une requête **PUT** est conçue pour modifier l'état de la ressource stockée à une certaine URI. Si l'URI de la requête fait référence à une ressource inexistante, celle-ci sera créée avec cette **URI**. **PUT** modifie donc l'état de la ressource, mais elle est idempotente : même si nous envoyons plusieurs fois la même requête **PUT**, l'état de la ressource finale restera inchangé.
- ✗ ***DELETE*** : Une requête **DELETE** supprime une ressource. La réponse à **DELETE** peut être un message d'état dans le corps de la réponse ou aucun code du tout. **DELETE** est idempotente, mais elle modifie évidemment l'état de la ressource.
- ✗ ***HEAD*** : ressemble à **GET** sauf que le serveur ne renvoie pas de corps dans sa réponse. **HEAD** permet par exemple de vérifier la validité d'un client ou la taille d'une entité sans avoir besoin de la transférer.
- ✗ ***TRACE*** : retrace la requête reçue.
- ✗ ***OPTION*** : est une demande d'information sur les options de communication disponibles pour la chaîne ***requête/réponse*** identifiée par l'URI. Cette méthode permet au client de connaître les options ***et/ou*** les exigences associées à une ressource, ou les possibilités d'un serveur sans demander d'action sur une ressource et sans récupérer aucune ressource.
- ✗ ***CONNECT*** : est utilisé avec un proxy pouvant se transformer dynamiquement en tunnel (une technique grâce à laquelle le protocole **HTTP** sert d'enveloppe à différents protocoles réseau).

✗ NÉGOCIATION DU CONTENU

La négociation de contenu est définie comme "le fait de choisir la meilleure représentation pour une réponse donnée lorsque plusieurs représentations sont disponibles". Les besoins, les souhaits et les capacités des clients varient.

La négociation du contenu utilise entre autres les en-têtes HTTP : ***Accept, Accept-Charset, Accept-Encoding, Accept-Language*** *et* ***User-Agent****. Pour obtenir, par exemple, la représentation* ***CSV*** *de la liste des livres sur Java publiés par Apress, l'application cliente (l'agent utilisateur) demandera* ***http://www.apress.com/books/catalog/java*** *avec un en-tête* ***Accept*** *initialisé à* ***text/csv****.*

- Vous pouvez aussi imaginer que, selon la valeur de l'en-tête **Accept-Language**, le contenu de ce document **CSV** pourrait être traduit par le serveur dans la langue correspondante.

✗ TYPES DE CONTENU

HTTP utilise des types de supports Intenet (*initialement appelés types MIME*) dans les en-têtes **Content-Type** et **Accept** afin de permettre un typage des données et une négociation de contenu ouverts et extensibles. Les types de support Internet sont divisés en cinq catégories : ***text, image, audio, video et application***. Ces types sont à leur tour divisés en sous-types (***text/plain, text/html, text/xhtml, etc***.). Voici quelques-uns des plus utilisés :

- **text/html** : **HTML** est utilisé par l'infrastructure d'information du World Wide Web depuis 1990 et sa spécification a été décrite dans plusieurs documents informels. Le type de support **text/html** a été initialement défini en 1995 par le groupe le groupe de traveil IETF HTML. Il permet d'envoyer et d'interpréter les pages web classiques.
- **text/plain** : Il s'agit du type de contenu par défaut car il est utilisé pour les messages textuels simples.
- **imagegif, image/jpeg, image/png** : Le type de support image exige la présence d'un dispositif d'affichage (un écran ou une imprimante graphique, apr exemple) permettant de visualiser l'information.
- **text/xml, application/xml** : Envoi et réception de document **XML**.
- **application/json** : **JSON** est un format textuel léger pour l'échange de données. Il est indépendant des langages de programmation.

✗ CODE D'ÉTAT

Un code **HTTP** est associé à chaque réponse. La spécification définit environ 60 codes d'états ; l'élément **Status-Code** est un entier de trois chiffres qui décrit le contexte d'une réponse et qui est intégéré dans l'enveloppe de celle-ci. Le premier chiffre indique l'une des cinq classes de réponses possibles :

- **1xx** : ***Information*** : La requête a été reçue et le traitement se poursuit.
- **2xx** : ***Succès*** : L'action a bien été reçue, comprise et acceptée.
- **3xx** : ***Redirection*** : Une autre action est requise pour que la requête s'effectue.
- **4xx** : ***Erreur du client*** : La requête contient des erreurs de syntaxe ou ne peut pas être exécutée.
- **5xx** : ***Erreur du serveur*** : Le serveur n'a pas réussi à exécuter une requête pourtant apparemment valide.

Voici quelques codes d'état que vous avez sûrement déjà dû rencontrer :

- ***200 OK*** : La requête a réussi. Le corps de l'entité, si elle en possède un, contient la représentation de la ressource.
- ***301 Moved Permanently*** : La ressource demandée a été affectée à une autre URI permanente et toute référence future à cette ressource devrait utiliser l'une des URI renvoyées.
- ***404 Not Found :*** Le serveur n'a rien trouvé qui corresponde à l'URL demandée.
- ***500 Internal Server Error*** : Le serveur s'est trouvé dans une situation inattendue qui l'a empêché de répondre à la requête.

✗ L'APPROCHE REST

Nous savons comment fonctionne le Web : pourquoi les services web devraient-ils se comporter différemment ? Après tout, ils échangent souvent uniquement des ressources bien identifiées, liées à d'autres au moyen de liens hypertextes. L'architecture du Web ayant prouvé sa tenue en charge au cours du temps, pourquoi réinventer la roue ?

Pour créer, modifier et supprimer une ressource livre, pourquoi ne pas utiliser ***les verbes classiques de HTTP*** *? Par exemple :*

- Utiliser **POST** sur des données (au format ***XML, JSON ou texte)*** afin de créer une ressource livre avec l'URI http://www.site.com/livres/. Le livre créé, la réponse renvoie l'URI de la nouvelle ressource http://www.apress.com/livres/123456.
- Utiliser **GET** pour lire la ressource (et les éventuels liens vers d'autres ressources à partir du corps de l'entité) à l'URI http://www.site.com/livres/123456.
- Utiliser **PUT** pour modifier la ressource à l'URI http://www.site.com/livres/123456.
- Utiliser **DELETE** pour supprimer la ressource à l'URI http://www.site.com/livres/123456.

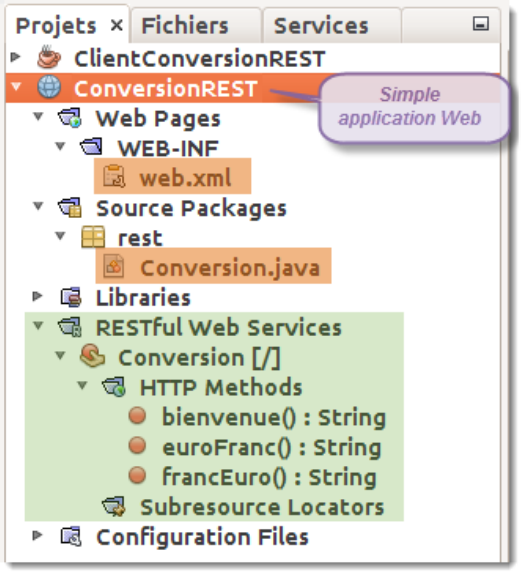
En se servant ainsi des ***verbes HTTP****, nous pouvons donc effectuer toutes les actions* ***CRUD*** *(Create, Read, Update, Delete) sur une ressource à l'image des bases de données.*

JAX-RS : JAVA API FOR RESTFUL WEB SERVICES

Vous pouvez vous demander à quoi ressemblera le code qui s'appuie sur des concepts d'aussi bas niveau que le protocole HTTP. En réalité, vous n'avez pas besoin d'écrire des requêtes HTTP ni de créer manuellement des réponses car **JAX-RS** est une API très élégante qui permet d'écrire une ressource à l'aide de quelques annotations seulement.

Je vous propose toute de simple de prendre un exemple extrêmement simple de service web REST qui utilise seulement la méthode GET. Ce service nous renvoie des textes non interprétés et qui réalise la conversion entre les euros et les francs.

- La première démarche consiste à utiliser une application Web au travers de laquelle nous allons annoter *une simple classe Java* (POJO) et renseigner le descripteur de déploiement **web.xml** qui va activer la servlet **ServletContainer** qui s'occupe d'implémenter la technologie **REST** (L'implémentation officielle de **JAX-RS** par Sun est **Jersey**).

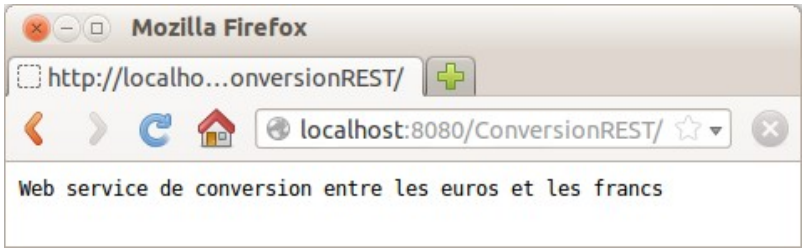


web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
        <servlet-name>ServletAdaptor</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>ServletAdaptor</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

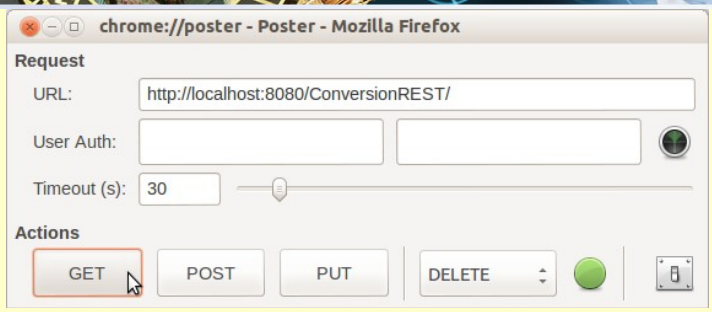
rest.Conversion.java

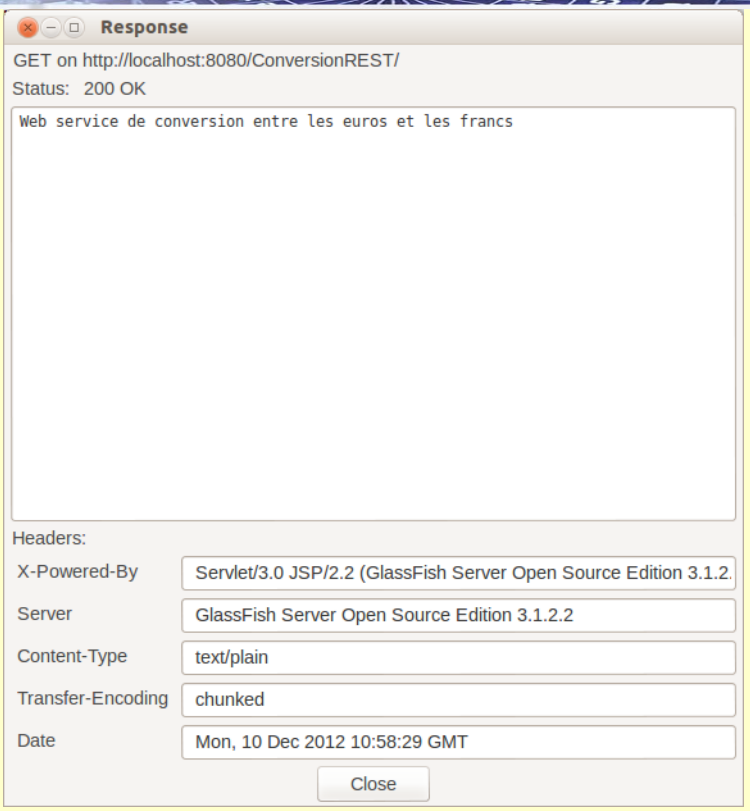
```
package rest;

import javax.ws.rs.*;

@Path("/")
@Produces("text/plain")
public class Conversion {
    private final double TAUX = 6.55957;

    @GET
    public String bienvenue() {
        return "Web service de conversion entre les euros et les francs";
    }

    @GET
    @Path("/{euro}€") // @Path("/franc/{euro}")
    public String euroFranc(@PathParam("euro") double euro) {
        return "" + (euro * TAUX);
    }

    @GET
    @Path("/{franc}F") // @Path("/euro/{franc}")
    public String francEuro(@PathParam("franc") double franc) {
        return "" + (franc / TAUX);
    }
}
```



- **Conversion** étant une classe Java annotée par **@Path**, la ressource sera hébergée à l'URI « **/** », la racine de l'application web.
- Les méthodes ***bienvenue(), euroFranc()*** et ***francEuro()*** sont elles-mêmes annotée par **@GET** afin d'indiquer qu'elles traiteront les requêtes **HTTP GET.**
- Ces méthodes produisent du texte. Le contenu est identifié par le type MIME « ***text/plain*** » grâce à l'annotation **@Produces**.
- Pour accéder aux ressources, il suffit d'un client **HTTP**, simple navigateur par exemple, pouvant envoyer une requête **GET** vers l'URL http://localhost:8080/ConversionREST/.

*Il est également possible d'utiliser un **client HTTP** qui permet de tester les services web de type **REST** en nous montrant les différentes en-têtes. Vous pouvez installer par exemple **le pluggin Poster** du navigateur Firefox (ou autre).*





*Le code précédent montre que le service **REST** n'implémente aucune interface et n'étend aucune classe : **@Path** est la seule annotation obligatoire pour transformer un POJO en service **REST**. **JAX-RS** utilisant la configuration par exception, un ensemble d'annotations permettent de modifier son comportement par défaut.*

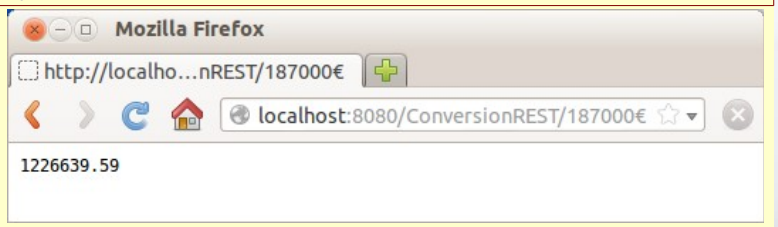
✗ Par nature, **JAX-RS** repose sur **HTTP** et dispose d'un ensemble de classes et d'annotations clairement définies pour gérer **HTTP** et les URI. Une ressource pouvant avoir plusieurs représentations, l'API permet de gérer un certain nombre de types de contenu et utilise **JAXB** pour sérialiser et désérialiser les représentations **XML** et **JSON** en objets.

✗ EXTRACTION DES PARAMÈTRES

Nous avons quelque fois besoin d'extraire des informations sur les URI et les requêtes lorsque nous les manipulons. **JAX-RS** fournit un ensemble d'annotation supplémentaires pour extraire les différents paramètres qu'une requête peut envoyer (**@PathParam, @QueryParam, @HeaderParam, et @DefaultValue**).

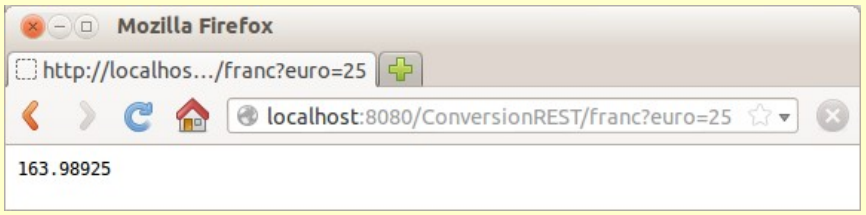
✗ **@PathParam** : Cette annotation permet d'extraire la valeur du paramètre d'une requête. Dans ce cas-là, vous intégrez dans la syntaxe de l'URI un nom de variable entouré d'accolades : ces variables seront ensuite évaluées à l'exécution. Le code suivant permet d'extraire la valeur en euro présente dans l'URI afin d'effectuer la conversion et de fournir la valeur numérique de l'équivalent en franc sous forme de simple chaîne de caractères (non interprété).

```
@GET
@Path("/{euro}€") // @Path("/franc/{euro})
public String euroFranc(@PathParam("euro") double euro) {
  return "" + (euro * TAUX);
}
```

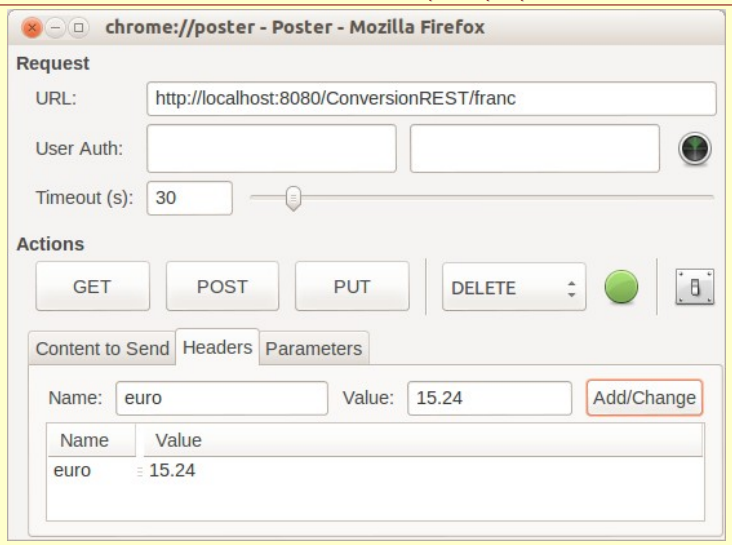


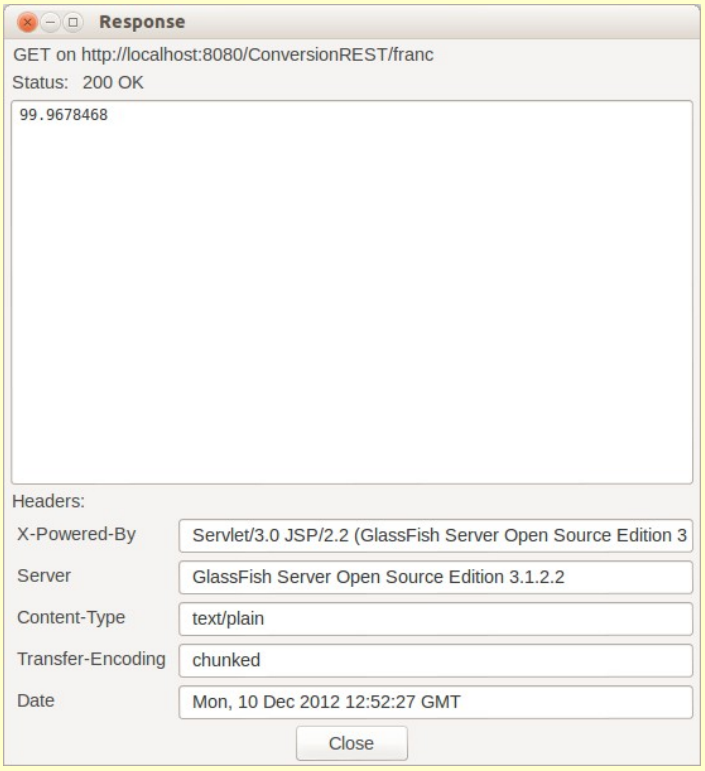
✗ **@QueryParam** : Cette annotation permet d'extraire la valeur d'un paramètre modèle d'une URI. Il s'agit ici d'une utilisation plus classique de la récupération de paramètres au moyen de la syntaxe usuelle prévue par la méthode **GET HTTP**.

```
@GET
@Path("franc")
public String euroFranc(@QueryParam("euro") double euro) {
  return "" + (euro * TAUX);
}
```



✗ **@HeaderParam** : Une autre méthode est plus liée aux détails internes de **HTTP**, ce que nous ne voyons pas directement dans les URI : notamment au travers des en-têtes. Cette annotation spécifique permet d'obtenir la valeur d'un en-tête, préfabriqué ou personnalisé :

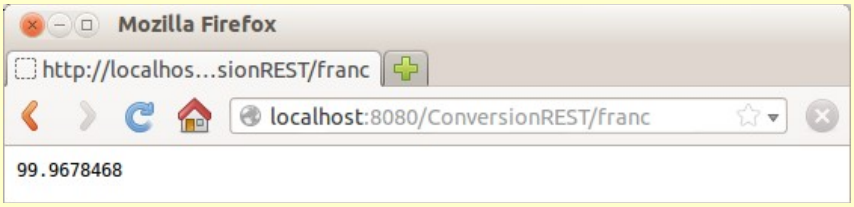





```
@GET
@Path("franc")
public String euroFranc(@HeaderParam("euro") double euro) {
   return "" + (euro * TAUX);
}
```

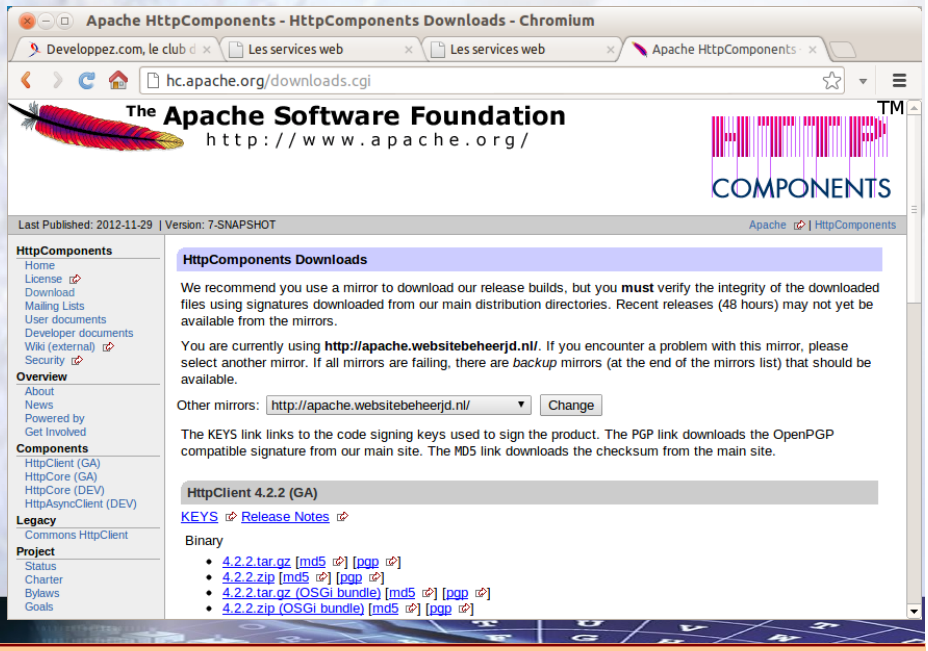
✗ **@DefaultValue** : Nous pouvons ajouter cette annotation à toutes celles que nous venons de découvrir pour définir une valeur par défaut pour le paramètre que nous attendons. Cette valeur sera utilisée si les métadonnées correspondantes sont absentes de la requête.

```
@GET
@Path("franc")
public String euroFranc(@DefaultValue("15.24") @QueryParam("euro") double euro) {
   return "" + (euro * TAUX);
}
```

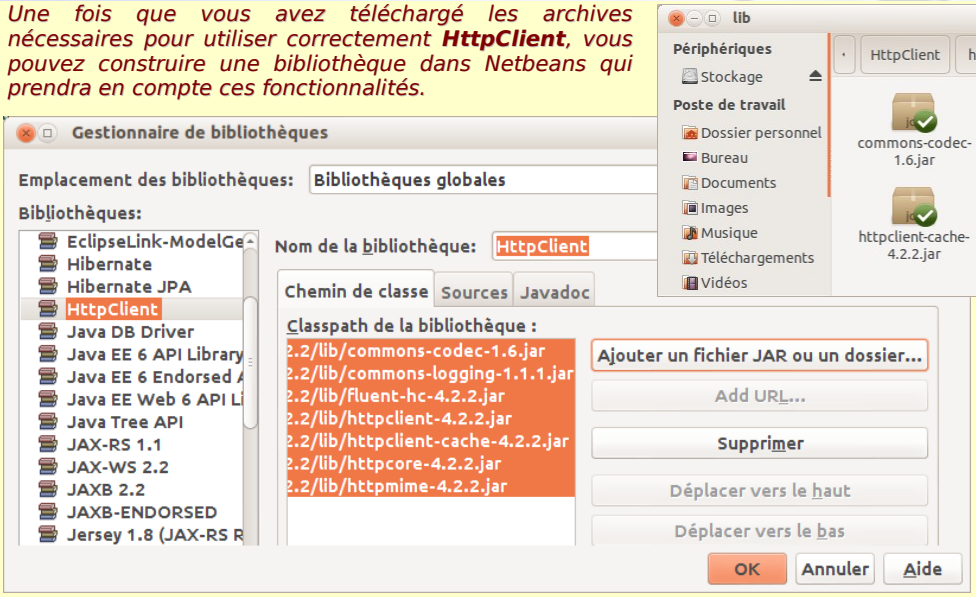


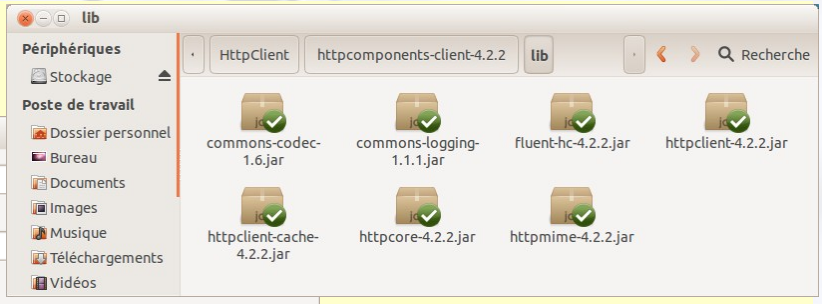
✗ UTILISATION DE LA CLASSE HTTPCLIENT (API D'APACHE)

Nous pouvons implémenter une application cliente Java classique qui permet de communiquer avec un service Web REST. Il suffit d'installer une API supplémentaire issue du projet Apache nommée **HttpClient** qui propose un certain nombre de fonctionnalités pour utiliser directement le protocole **HTTP** dans son ensemble.



*Une fois que vous avez téléchargé les archives nécessaires pour utiliser correctement **HttpClient**, vous pouvez construire une bibliothèque dans Netbeans qui prendra en compte ces fonctionnalités.*





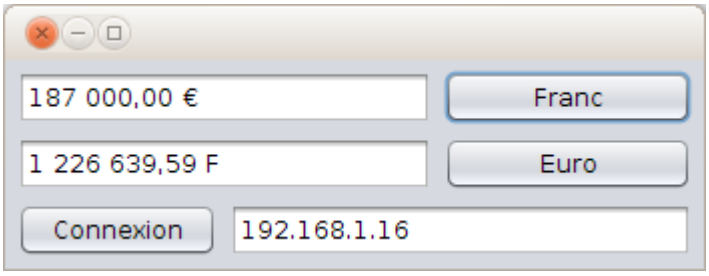
*Quel que soit le type de requête, **GET, POST, PUT, DELETE, HEAD et OPTION**, l'utilisation de ce client suit le schéma suivant :*

- ✗ Création d'une instance de la classe **DefaultHttpClient** qui implémente l'interface **HttpClient**.
- ✗ Création d'une instance d'un objet représentant la requête qui spécialisera notre client.
- ✗ Réglage des propriétés de cette requête.
- ✗ Exécution de la requête grâce à l'instance de type **HttpClient** obtenue auparavant.
- ✗ Analyse et traitement de la réponse.

*Au delà de l'utilisation de la classe **DefaultHttpClient** et de son interface **HttpClient**, vous devez prendre une classe correspond à la requête **HTTP** souhaitée, ici la requête **GET**. La classe correspondante se nomme tout simplement **HttpGet**. Pour récupérer la réponse à la requête choisissez cette fois-ci la classe spécialisée **HttpResponse**.*

✗ MISE ŒUVRE DE L'APPLICATION CLIENTE JAVA

rest.Conversion.java




```
package rest;

import java.io.IOException;
import java.util.*;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;

public class Convertir {
    private String localisation;

    public void seConnecter(String adresse) throws IOException {
        localisation = "http://"+adresse+":8080/ConversionREST/";
        // Uniquement pour tester la connexion et savoir si le service est opérationnel
        HttpClient client  = new DefaultHttpClient();
        HttpGet requête = new HttpGet(localisation);
        HttpResponse réponse = client.execute(requête);
    }

    public double francEuro(double franc) throws IOException {  return soumettre(franc+"F");  }
    public double euroFranc(double euro) throws IOException{  return soumettre(euro+"€");  }

    private double soumettre(String URI) throws IOException {
        HttpClient client  = new DefaultHttpClient();
        HttpGet requête = new HttpGet(localisation+URI);
        HttpResponse réponse = client.execute(requête);
        if (réponse.getStatusLine().getStatusCode() == 200) {
            Scanner valeur = new Scanner(réponse.getEntity().getContent());
            valeur.useLocale(Locale.US);
            return valeur.nextDouble();
        }
        return 0.0;
    }
}
```

rest.Principal.java

```
package rest;

import java.io.IOException;
import javax.swing.JOptionPane;

public class Principal extends javax.swing.JFrame {
    private Convertir service = new Convertir();

    public Principal() {  initComponents();  }

    private void initComponents() { ... }// </editor-fold>

    private void pourFrancActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            Number valeur = (Number) euro.getValue();
            franc.setValue(service.euroFranc(valeur.doubleValue()));
        }
        catch (IOException ex) {
            JOptionPane.showMessageDialog(this, "Impossible de communiquer avec le service");
        }
    }

    private void pourEuroActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            Number valeur = (Number) franc.getValue();
            euro.setValue(service.francEuro(valeur.doubleValue()));
        }
        catch (IOException ex) {
            JOptionPane.showMessageDialog(this, "Impossible de communiquer avec le service");
        }
    }

    private void connexionActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            service.seConnecter(adresse.getText());
            JOptionPane.showMessageDialog(this, "Service opérationnel...");
        }
        catch (IOException ex) {
            JOptionPane.showMessageDialog(this, "Impossible de communiquer avec le service");
        }
    }

    public static void main(String args[]) { ... }

    private javax.swing.JTextField adresse;
    private javax.swing.JButton connexion;
    private javax.swing.JFormattedTextField euro;
    private javax.swing.JFormattedTextField franc;
    private javax.swing.JButton pourEuro;
    private javax.swing.JButton pourFranc;
}
```

UTILISATION DE TOUTES LES MÉTHODES HTTP

Maintenant que nous avons pris connaissance de l'utilisation de la méthode **GET**, au travers d'un projet d'archivage de photos, nous allons utiliser les autres méthodes **HTTP**, suivant les différents cas d'utilisation.

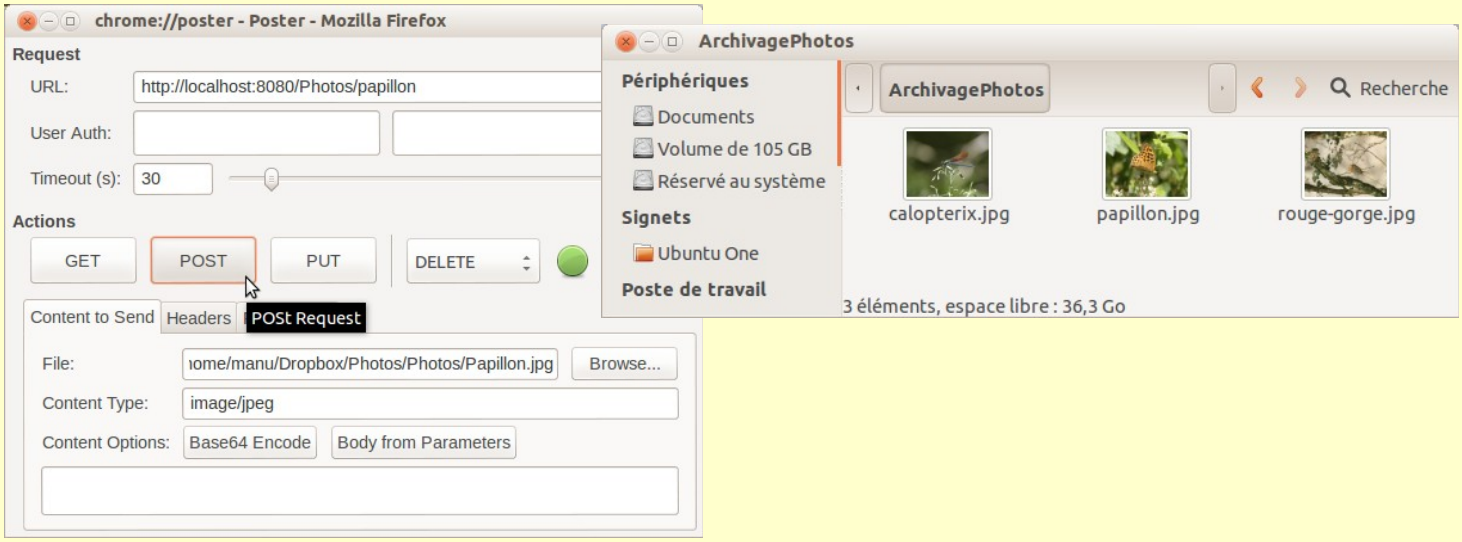
service.Archivage.java

```
package service;

import java.io.*;
import javax.ws.rs.*;

@Path("/")
public class Archivage {
    private final String répertoire = "/home/btsiris/ArchivagePhotos/";

    @GET
    @Path("{nomFichier}")
    @Produces("image/jpeg")
    public InputStream restituer(@PathParam("nomFichier") String nom) throws FileNotFoundException {
        return new FileInputStream(répertoire+nom+".jpg");
    }
```

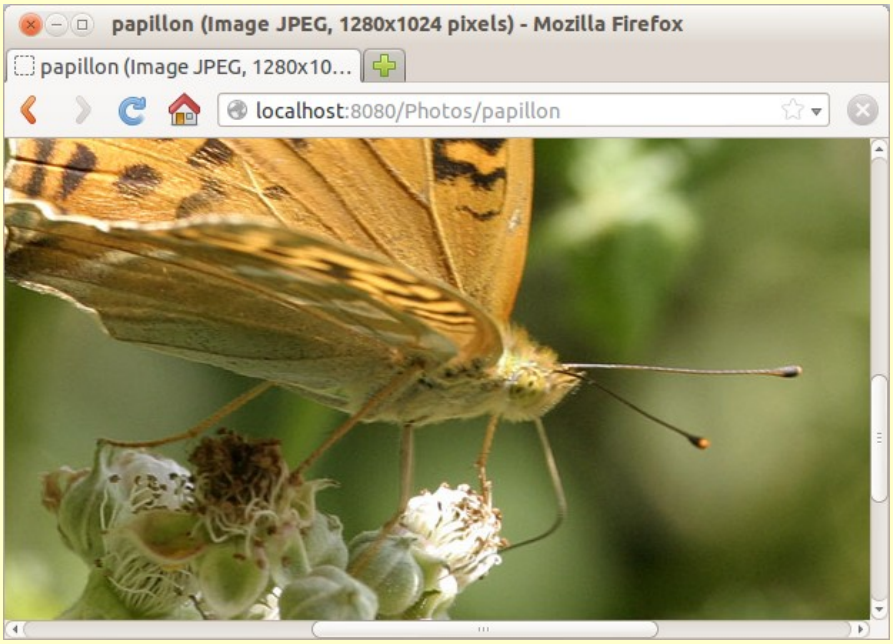
```
    @GET
    @Produces("text/plain")
    public String listePhotos() {
        String[] liste = new File(répertoire).list();
        StringBuilder chaîne = new StringBuilder();
        for (String nom : liste) chaîne.append(nom.split(".jpg")[0] +"\n");
        return chaîne.toString();
    }

    @POST
    @Path("{nomFichier}")
    @Consumes("image/jpeg")
    public void stocker(@PathParam("nomFichier") String nom, InputStream flux) throws IOException {
        byte[] octets = lireOctets(flux);
        FileOutputStream fichier = new FileOutputStream(répertoire+nom+".jpg");
        fichier.write(octets);
        fichier.close();
    }

    @PUT
    @Path("change")
    public void changerNom(@QueryParam("ancien") String ancien, @QueryParam("nouveau") String nouveau) {
        new File(répertoire+ancien+".jpg").renameTo(new File(répertoire+nouveau+".jpg"));
    }

    @DELETE
    @Path("{nomFichier}")
    public void supprimer(@PathParam("nomFichier") String nom) {
        new File(répertoire+nom+".jpg").delete();
    }

    private byte[] lireOctets(InputStream stream) throws IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        byte[] buffer = new byte[1024]; int octetsLus = 0;
        do {
            octetsLus = stream.read(buffer);
            if (octetsLus > 0) { baos.write(buffer, 0, octetsLus); }
        }
        while (octetsLus > -1);
        return baos.toByteArray();
    }
}
```

✗ **POST** : Nous permet d'archiver une photo identifiée sur le disque dur du serveur. Pour cela, vous devez spécifier le nom de votre fichier image à la fin de votre URI. Remarquez la présence d'un paramètre de type **InputStream** qui va nous permettre de récupérer le flot d'octets issu du fichier image envoyé après l'en-tête du protocole **HTTP**. Le type **MIME** est « ***image/jpeg*** ». Ainsi le document proposé est directement le fichier image lui-même. Remarquez également la présence de l'annotation **@Consumes** en lieu et place de **@Produces**, puisque cette fois-ci c'est le document qui est reçu par le service.

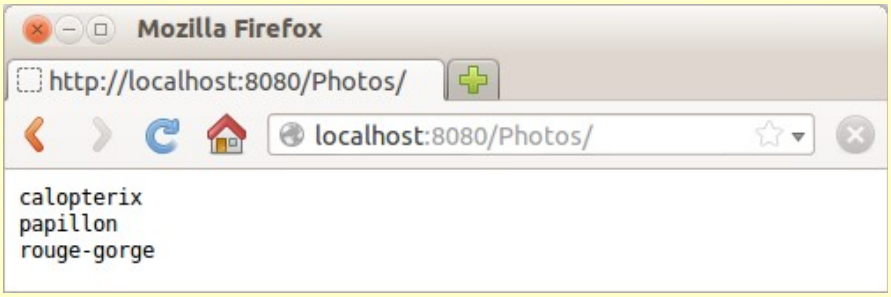
✗



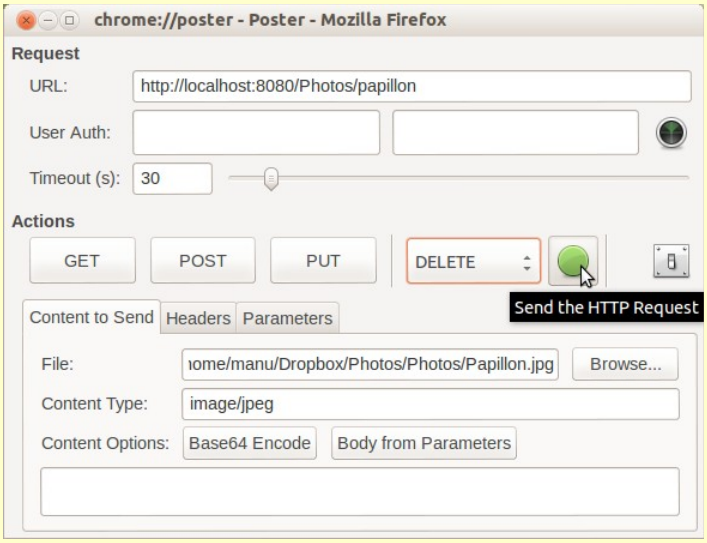
✗ **GET** : Nous permet de récupérer une photo en particulier en désignant le nom du fichier à la suite de l'URI. La méthode ***restituer()*** s'occupe de cela et renvoie, nous nous en doutons, un **InputStream**.



GET : Deuxième méthode surchargée qui nous procure la liste de toutes les photos présentes sur le disque dur du serveur actuellement.



✗ **DELETE** : Nous permet d'enlever une photo du disque dur du serveur.



✗

✗ **PUT** : Nous permet de changer le nom du fichier sur le disque dur du serveur.

✗