

Tutoriel de prise en main : Restlet

Jonathan Lejeune



Objectifs

L'objectif de ce document est de compléter le cours sur les Web-services qui vous a été fourni pour vous familiariser avec l'utilisation d'un outil pour programmer et déployer des services RESTful en codant un petit exemple. Ce document est une prise en main rapide de Restlet. Il est bien sûr conseillé de lire de la documentation complémentaire pour tout approfondissement.

Aperçu de Restlet

Restlet est une plateforme développée par l'entreprise française Talend qui est un acteur important dans le monde informatique à l'international. Cette plateforme possède les avantages suivants :

- elle est écrite en java et elle est open-source
- elle dispose de plusieurs éditions : une édition pour les plateformes mobiles Android, une édition pour le cloud de Google (App Engine), une édition pour les serveurs d'application JavaEE et enfin une édition JavaSE pour les applications java classiques. C'est sur cette édition JavaSE que nous nous baserons, car elle ne nécessite pas d'installation lourde pour l'utiliser (un simple jdk de base suffit).
- son API est basée sur l'assemblage de composants : elle est relativement intuitive et simple à utiliser
- elle supporte la plupart des standards du Web (HTTP, SMTP, XML, JSON ...)
- elle offre des extensions pouvant s'intégrer avec des Servlet et des serveurs web reconnus (Jetty) et qui peuvent planter des API standards comme JAX-RS.

Prérequis

- Avoir installé le JDK 8 (ou supérieur) sur votre machine.
- Avoir un IDE pour Java (ici nous utiliserons Eclipse).
- Avoir lu et compris le cours associé.

Tuto 1 – Installation de Restlet dans Eclipse

Étape 1

Téléchargez l'édition JavaSE de Restlet sur <https://restlet.talend.com/downloads/current/> et désarchivez-la dans un dossier de votre choix.

Étape 2

Afin de regrouper l'ensemble des jars dans le même dossier, aller dans le dossier *lib* du dossier téléchargé et exécuter le script suivant :

```

#!/bin/bash

path_target=$(dirname $0)
cd $path_target
path_target=$(pwd)

for f in * ; do
    if ! [ -d $f ] ; then
        continue
    fi
    cd $f
    for jar in *.jar ; do
        rm -f "$path_target/$jar"
        ln -s "$PWD/$jar" "$path_target/$jar"
    done
    cd $path_target
done

```

Ce script a pour effet de créer des liens symboliques vers les jars présents dans les sous-dossiers de *lib*

Étape 3

Ouvrir Eclipse et créer un nouveau projet Java (nommons-le par exemple TutoRestLet).

Étape 4

Pour compiler et exécuter vos programmes Restlet dans Eclipse, vous devez d'abord déclarer une nouvelle bibliothèque Java qui contiendra les différents fichiers jar de Restlet. Pour cela :

- Aller dans *Window* → *Preferences* → Dérouler l'onglet *Java* → Dérouler l'onglet *Build Path* → cliquer sur *User Librairies*.
- Cliquer sur le bouton *New...* à droite et renseigner le nom de la nouvelle bibliothèque *restlet* en suffixant par la version téléchargée. Cliquer sur le bouton *OK*.
- Sélectionner la bibliothèque et cliquer sur le bouton *Add External Jars...* à droite.
- Sélectionner tous les fichiers jar se trouvant dans le sous-dossier *lib* de votre dossier d'installation restlet ainsi que tous les liens symboliques créés à l'étape 2
- (optionnel) Il peut être utile de relier les sources java de la bibliothèque à certains jars centraux (pour comprendre un logiciel, rien de mieux de lire son code). Ici vous pouvez le faire pour le jar *org.restlet.jar* en le déroulant et en cliquant sur la ligne *Source attachment*. Cliquer sur *Edit...* et renseigner une archive zip ou un dossier contenant les fichiers sources.

Étape 5

Pour ajouter cette bibliothèque au projet Eclipse :

- clic droit sur le projet → *Build Path* → *Add Librairies...*
- Choisir *User Library* → *Next >*
- Cocher la bibliothèque précédemment créée → *Finish*

Étape 6

Créez la classe suivante et vérifizr que la compilation se passe bien (absence de croix rouge) :

```

package srcts.restlet.hello;

import org.restlet.Request;
import org.restlet.Response;

```

```

import org.restlet.Restlet;
import org.restlet.Server;
import org.restlet.data.MediaType;
import org.restlet.data.Protocol;

public class HelloBasic {

    public static void main(String[] args) throws Exception {
        Server s = new Server(Protocol.HTTP, 8585, new Restlet() {
            @Override
            public void handle(Request request, Response response) {
                response.setEntity("Hello !!\n", MediaType.TEXT_PLAIN);
            }
        });
        s.start();
    }
}

```

Étape 7

Lancer le programme précédent, ouvrir un terminal, taper la commande `curl http://localhost:8585/` et vérifier que le message de retour s'affiche bien (il est également possible d'utiliser un navigateur).

Étape 8

Eclipse possède également un navigateur intégré. Pour l'utiliser aller dans *Window → Show View → Other....* Dérouler la partie *General* et cliquer sur *Internal Web Browser*. Ceci peut être utile pour tester les programmes sans changer de fenêtre.

Présentation de l'API Restlet

La principale problématique pratique d'une telle plateforme est la capacité à comprendre une requête HTTP et appeler le bon code pour traiter la requête et envoyer une réponse au client. Le déploiement côté serveur se base sur la construction et l'assemblage de différents composants appelés des **restlet**. Cet assemblage va ainsi décrire la cartographie permettant le cheminement d'une requête d'un composant de départ (gérant une socket d'écoute) vers un composant d'arrivée (gérant la ressource de la requête) qui effectuera le traitement.

Aperçu logiciel

D'un point de vue Java, la figure 1 montre un diagramme de classes des différents types de composants offerts par la plateforme.

On peut remarquer que cette architecture logicielle suit un pattern composite permettant d'organiser les objets de manière arborescente :

- les restlets composites qui contiennent une ou plusieurs références vers d'autres restlet et qui ont le rôle de contrôler le cheminement des requêtes :
 - ◊ les **Servers** et les **Filters** possèdent une référence sur un restlet **next**
 - ◊ les **Routers** possèdent une liste de références de servlets
 - ◊ les **Applications** possèdent deux références vers deux servlets racines
 - ◊ les **Components** possèdent des références sur des **VirtualHosts**, des **Servers** et des **Clients**
- les restlets terminaux qui permettent de traiter une requête
 - ◊ les **Finders** pour les ressources hébergées sur la machine hôte

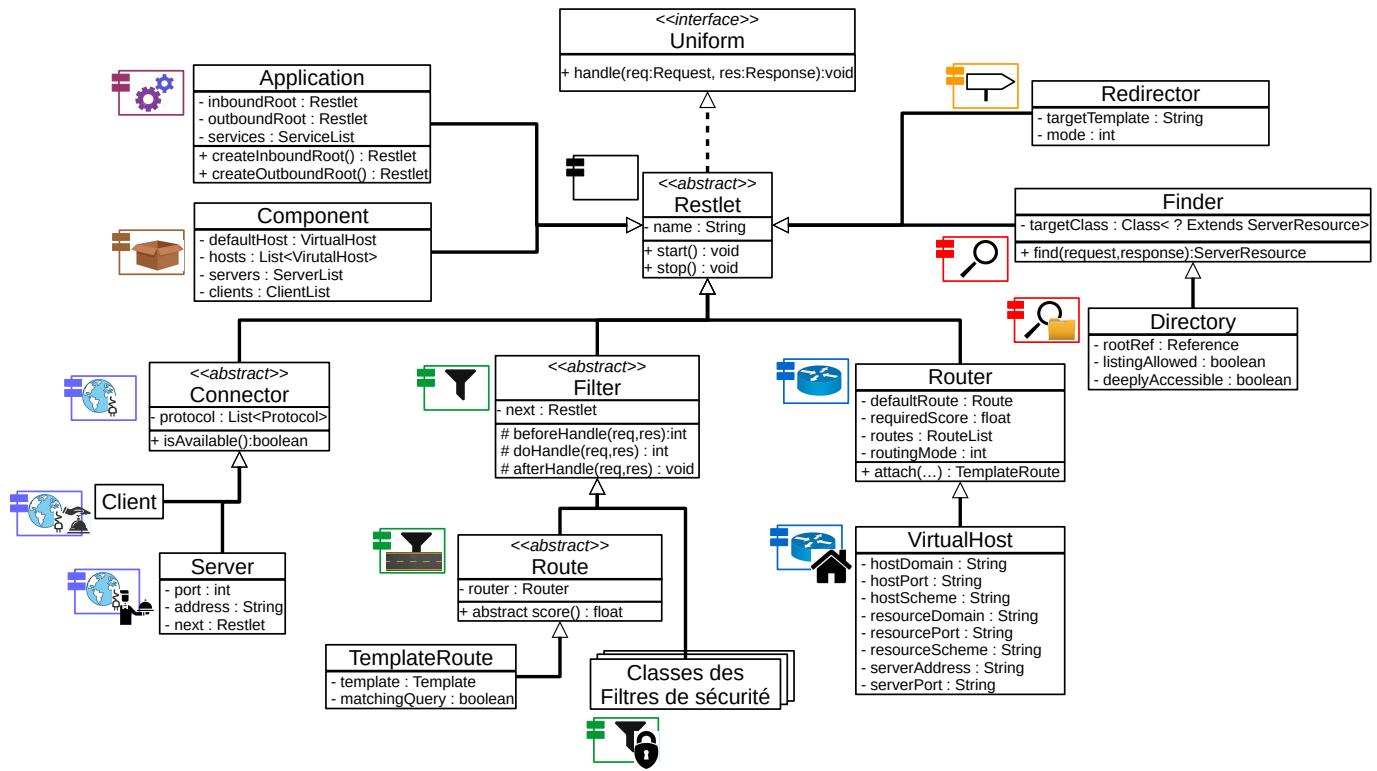


FIGURE 1 – Diagramme de classe des différents types de restlet

- ◊ les **Redirectors** pour la retransmission de requêtes vers un autre URI (serveurs intermédiaires)
 - ◊ les **Clients** pour gérer les connexions clientes vers d'autres web services

La classe Restlet

La classe Restlet est la classe générique de tout composant d'une plateforme Restlet. Tout restlet propose 3 opérations :

- une méthode de démarrage `start` permettant d'activer et de déployer le restlet (ainsi que ses sous-restlet si c'est un composite).
 - une méthode d'arrêt `stop` permettant de désactiver le restlet (ainsi que ses sous-restlets si c'est un composite).
 - la méthode `handle` héritée de l'interface `Uniform` permettant de traiter une requête HTTP. Elle possède deux paramètres : un objet `Request` contenant les informations de la requête et un objet `Response` contenant les éléments de la réponse qu'il faut éditer lors du traitement. On remarquera que ce type d'interface est similaire aux Servlet Java. C'est cette méthode qui est appelée de manière récursive entre les différents restlets lors du cheminement d'une requête. Ce mécanisme suit un design pattern reconnu en génie logiciel appelé *Chain of Responsibility*.

Les connecteurs

Les connecteurs permettent de faire le lien entre la JVM et l'extérieur via le réseau selon un protocole applicatif standard (HTTP, HTTPS, FTP, ...). Un connecteur peut supporter plusieurs protocoles. Son principal rôle est la traduction entre le format réseau des messages (tel qu'il a été défini dans le protocole) et des objets java qui décrivent les requêtes et les réponses. Il existe deux types de connecteurs :

- les clients implantés par la classe `Client` qui permettent de gérer un pool de connexions vers des serveurs distants selon les protocoles que le connecteur peut supporter.
- les serveurs implantés par la classe `Server` qui permettent de définir une socket d'écoute sur un port et une interface IP donnée sur la machine hôte. C'est ce type de restlet qui est le premier maillon de la chaîne de transmission et initiant ainsi le cheminement de la requête/réponse via son lien `next`.

Les connecteurs sont multi-threadés à l'aide de pool de threads. Ceci signifie, notamment pour les serveurs, qu'**un thread est dédié à chaque requête entrante**.

Les filtres

Les filtres sont des restlets transitoires et possèdent une référence vers un restlet suivant (attribut `next`). Leur rôle permet de contrôler le cheminement d'une requête :

- en bloquant ou en laissant passer la requête en fonction d'une condition. Dans le cas d'un blocage, le filtre met fin au cheminement et paramètre la réponse en fonction de l'erreur détectée. Ce type de filtre peut être utilisé pour des aspects de sécurité (ex : pas les bons droit d'accès à la ressource, adresse IP du client blacklistée ...). Nous ne traiterons pas de l'aspect sécurité dans ce tutoriel, mais il est important de savoir qu'il existe.
- en ajoutant des traitements intermédiaires comme par exemple la production de logs (`LogFilter`) ou le décodage/l'encodage de données

L'implantation de la méthode générique `handle` d'un filtre fait appel à 3 méthodes dans cet ordre `beforeHandle`, `doHandle` et `afterHandle`. La méthode `beforeHandle` permet d'avoir un contrôle sur la requête avant l'appel au restlet suivant. Elle retourne un entier dont la valeur indique si la requête peut continuer ou pas (`CONTINUE`, `SKIP` ou `STOP`). La méthode `afterHandle` est appelée lors du retour de la réponse ou bien directement après la méthode `beforeHandle` si cette dernière a renvoyé `SKIP`.

Les routes

Les routes sont des filtres non bloquants et sont associées à une étiquette et à un restlet de type routeur (cf. plus loin). Pour traverser une route, il est nécessaire que l'URI de la ressource demandée corresponde à l'étiquette de la route. Une route offre la méthode `score` qui renvoie un nombre compris entre 0 et 1. Plus la valeur de ce score est élevée, plus l'URI correspond à l'étiquette de la route. Une fois qu'une route a été choisie (par le routeur associé), la partie de l'URI matchant l'étiquette de la route ne sera plus prise en compte dans la suite du cheminement. Les types de routes les plus utilisés sont les `TemplateRoute` qui permettent de router les requêtes une fois que les parties `scheme` et `authority` de l'URI ont été traitées. Le routage se fait donc en fonction du `path` de leur URI (et éventuellement de la partie `query` si le booléen `matchingQuery` est vrai). L'étiquette de ce type de route est un pattern de chaîne de caractères. Le score est alors la proportion de caractères de l'URI restant à traiter dans le cheminement qui correspondent au template de la route. Le choix de la route est cependant de la responsabilité du routeur associé.

Considérons par exemple une route `A` étiquetée par le template `/toile` et une route `B` étiqueté par `/toi`. Si un URI est composé de la chaîne `/toile/verte` restante à traiter, il aura un plus gros score sur la route **A** que sur la route **B**. En revanche si une URI est composé de la chaîne `/toi/moi`, il aura un score nul sur `A` car il n'y a pas de correspondance possible, mais aura un score plein pour la route **B**.

On notera que la plupart du temps une étiquette correspond à un ou plusieurs étages complets du chemin de l'URI.

Les routeurs

Un routeur est un servlet permettant de choisir la suite du cheminement parmi un ensemble de routes. Le routeur possède un score minimal requis pour choisir une route (attribut `requiredScore` avec une valeur défaut à 0.5). Il existe différentes politiques de choix de la route à emprunter :

- le mode `BEST_MATCH` : le routeur choisit la route qui a le meilleur score parmi les routes qui ont le score minimal requis
- le mode `FIRST_MATCH` (Mode par défaut) : le routeur choisit la première route qui a le score minimal requis en partant du début de la liste
- le mode `LAST_MATCH` : le routeur choisit la première route qui a le score minimal requis en partant de la fin de la liste
- le mode `NEXT_MATCH` : le routeur choisit la première route qui a le score minimal requis en suivant une politique de round-robin
- le mode `RANDOM` : le routeur choisit au hasard une route qui a le score minimal requis

Pour associer une nouvelle route à un routeur, il faut appeler la méthode d'instance `attach` qui crée une nouvelle `TemplateRoute`. Cette méthode peut prendre trois paramètres :

- obligatoirement une cible qui peut être :
 - ◊ soit un restlet quelconque (type `Restlet`)
 - ◊ soit une classe (réflexivité java) qui étend le type `ServerResource` (cf. plus loin). Ceci a pour conséquence de créer un restlet cible de type `Finder` (cf. plus loin).
- optionnellement une étiquette (valeur par défaut = chaîne vide)
- optionnellement un entier précisant une politique de correspondance. Il existe deux politiques de correspondance :
 - ◊ `MODE_EQUALS` : l'URI doit parfaitement correspondre aux caractères de l'étiquette
 - ◊ `MODE_STARTS_WITH` : l'URI doit commencer par les caractères de l'étiquette

En pratique il n'est donc pas nécessaire d'instancier directement les routes car c'est la méthode `attach` qui le fait pour nous et qui fait la liaison entre son point de départ (le routeur) et son point d'arrivée (sa cible renseignée en argument).

On note l'existence de routeurs de type `VirtualHost` qui permettent de router les URI en fonction de leur *scheme* et de leur *authority*. Ce type de routeur se trouve donc en amont du cheminement de la requête. Un `VirtualHost` permet d'avoir plusieurs authority sur la même interface réseau ce qui donne ainsi la possibilité d'avoir plusieurs serveurs virtuels sur la même machine.

Dans tous les cas, si un routeur ne trouve aucune route alors le cheminement de la requête s'arrête et une réponse est envoyée au client en précisant que la ressource est introuvable (en HTTP, la fameuse erreur 404 Not Found).

Les restlets de fin de cheminement

Dans cette partie, nous allons décrire les restlets qui marquent la fin du cheminement d'une requête. Ces restlets sont donc des composants qui gèrent ou représentent la ressource demandée.

Les Finders

Un restlet de type finder possède une référence vers une sous-classe de la classe `ServerResource`. Une implantation de la classe `ServerResource` permet de définir le traitement à effectuer pour une ressource. Lorsqu'une requête aboutit sur un finder, ce dernier instancie une nouvelle instance de la classe `ServerResource` cible et affecte cette instance pour la requête en question. Ce processus permet de respecter un mode stateless (contrainte de REST) et empêche à toute requête de maintenir un état sur la ressource qui lui est propre. Les finders de type `Directory` permettent de faire correspondre la ressource avec un répertoire du système de fichiers de la machine hôte. S'il reste des caractères à traiter dans l'URI en atteignant ce type de restlet, alors la plateforme y cherchera une correspondance

dans les chemins de la sous arborescence du dossier. Si il n'y a plus de caractère à traiter dans l'URI alors la réponse sera composée de la liste des fichiers du répertoire si l'attribut `listingAllowed` est positionné à vrai.

Les redirecteur

Ce type de restlet permet de rediriger la requête vers un autre URI cible. Ce type de restlet sert principalement de proxy ou de reverse-proxy

Les applications

Les applications permettent de regrouper logiquement un ensemble de Restlets dans un contexte. Ceci permet de cloisonner les différents chemins de traitement des requêtes. Une application doit référencer un restlet racine `inBoundRoot` qui est le point d'entrée dans l'application. Une application offre également des fonctionnalités (services) qui peuvent être utiles au cheminement et au traitement de la requête. On peut citer par exemple le service `Converter` qui permet de faire des traductions entre plusieurs représentation (XML, Text, JSON, ...). Dans la plateforme Restlet, l'exécution d'une requête sur une ressource doit se faire obligatoirement dans la cadre d'une application. Il est néanmoins possible de ne pas déclarer d'application dans le code de déploiement des restlets. Cependant, en l'absence d'application, la plateforme créera systématiquement une application temporaire pour chaque requête ce qui alourdira le traitement.

Les components

Un component peut être vu comme un conteneur englobant tous les restlets du serveur (ou de la JVM). Il fait ainsi office de conteneur global et permet de respecter une logique dans le chemin de traitement d'une requête. En partant de l'amont vers l'aval, un component possède :

- une liste de connecteurs serveurs permettant de recevoir les requêtes provenant du réseau.
- un routeur interne `ServerRouter` (objet implicitement existant) permettant de router les requêtes vers le bon serveur virtuel afin de décoder la partie `scheme` et `authority` de l'URI des requêtes.
- une liste de `VirtualHosts`. Un component comporte au moins un serveur virtuel par défaut à sa création.
- un routeur interne `ClientRouter` (objet implicitement existant) permettant de router les requêtes sortantes des restlets du component vers le bon connecteur client (en fonction du protocole applicatif utilisé).
- une liste de connecteurs clients permettant à ses restlets d'envoyer des requêtes à d'autres services distants

Assemblage type d'un serveur

La figure 2 schématisé un exemple d'assemblage de restlet.

Dans cet exemple on remarque qu'il existe deux sockets d'écoute sur les ports 80 et 8080 pour le protocole HTTP ainsi que 3 hôtes virtuels. Les URI admissibles doivent donc tous avoir commencer par :

- `http://localhost/` ou `http://localhost:80/` ou `http://localhost:8080/` (assuré par l'hôte virtuel par défaut)
- `http://foo.com/` ou `http://foo.com:80/` ou `http://foo.com:8080/`
- `http://foo.fr/` ou `http://foo.fr:80/` ou `http://foo.fr:8080/`

En supposant que la politique de matching des routes sont tous en mode `STARTS_WITH`, on remarque :

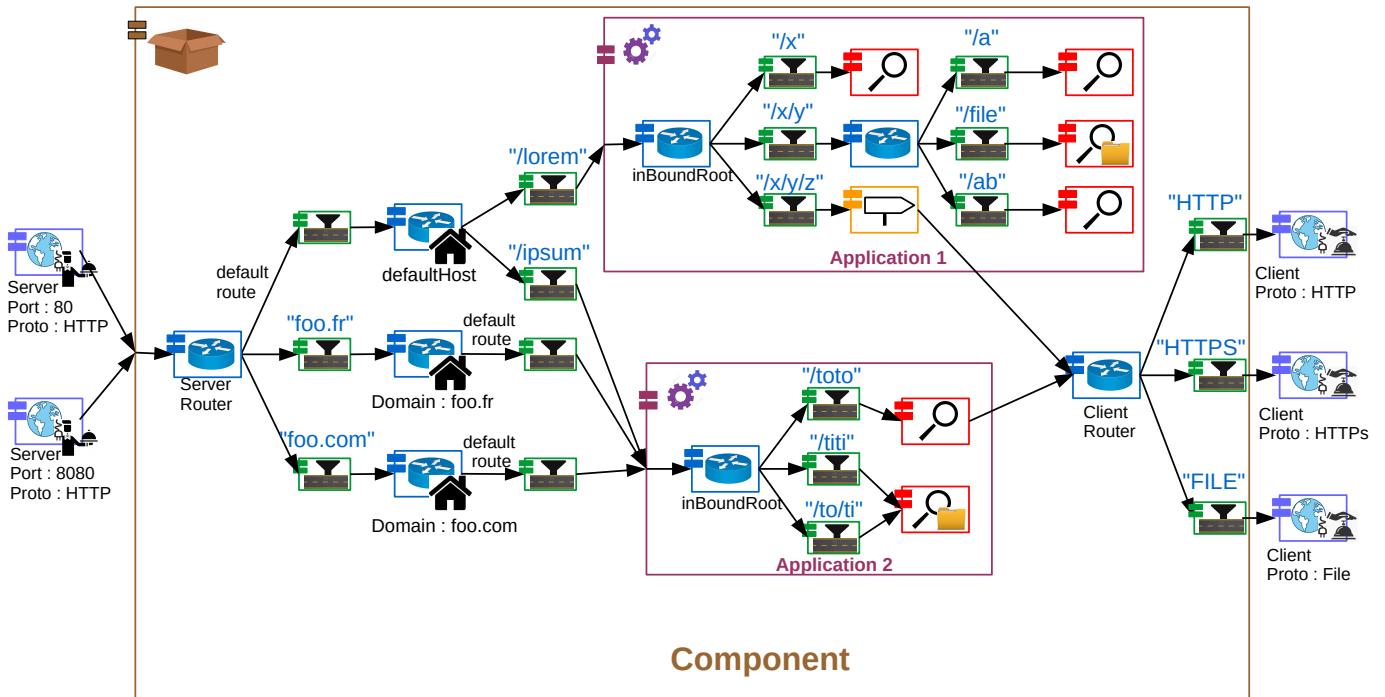


FIGURE 2 – Exemple type d’assemblage de restlets

- que toute requête transitant par l'hôte virtuel pourront aller soit dans l'application 1 si leur path commence par `"/lorem"` soit dans l'application 2 si leur path commence par `"/ipsum"`. Tout URI concernant l'application 1 doit donc commencer par le domaine de l'hôte virtuel par défaut et avoir un path préfixé par `"/lorem"`
- que toute requête transitant par les deux autres hôtes virtuels aboutiront sur l'application 2 et il n'y a aucune contrainte de préfixe sur le path
- que dans l'application 2 les paths `"/titi"` et `"/to/ti"` désignent la même ressource
- qu'une route peut regrouper plusieurs niveaux de hiérarchie du path (c'est le cas par exemple pour les routes `"/x/y"`, `"/x/y/z"` de l'application 1 et la route `"/to/ti"` de l'application 2)
- qu'un routeur ne correspond pas forcément à un étage du path de l'URI : dans l'application 1 la ressource `"/x/y/z"` n'est pas gérée par le même routeur que les ressources `"/x/y/a"`, `"/x/y/file"`, , `"/x/y/ab"`.
- que les applications sont transparentes au niveau des URIs (et donc des clients) et n'ont juste qu'une logique interne propre au serveur

Définir le code de traitement d'une requête

Comme nous l'avons vu précédemment, une requête doit aboutir sur un restlet terminal pour qu'elle soit traitée. Pour cela il existe deux solutions.

Solution 1

À l'image du programme `HelloBasic` donné plus haut, la première solution consiste à définir une classe qui étend `Restlet` et redéfinir la méthode `handle`. Il faut ensuite instancier cette classe une seule fois et relier l'instance à un restlet existant (la plupart du temps un routeur). Cette solution offre une API peu évoluée pour traiter les requêtes. En effet, toute requête sur une même ressource aboutira sur la même méthode `handle` du même objet. La plateforme ne se soucie donc pas de la méthode HTTP de la requête ou bien de la représentation de la ressource souhaitée par le client (négociation de contenu). C'est donc au programmeur de gérer manuellement la méthode et éventuellement la

conversion entre les différentes représentation de données. De plus, comme évoquée précédemment, cette méthode n'est pas stateless (donc contraire à la philosophie REST) car on maintient un objet global pour toutes les requêtes.

Solution 2

C'est la solution la plus courante. Elle consiste à créer une classe qui étend la classe `ServerResource` et d'y définir des méthodes annotées. Pour déployer ce type de ressource il faut passer en paramètre de la méthode `attach` d'un routeur, la classe en question. Ceci aura pour effet de construire un restlet de type finder qui instanciera à chaque nouvelle requête cette classe. Une fois instanciée, la classe finder a la responsabilité de :

- chercher la bonne méthode de la classe à appeler en fonction de leurs annotations et de la méthode de la requête HTTP.
- contrôler la négociation de contenu en paramétrant la réponse correctement en y insérant la représentation des données (type MIME) la plus adéquate.

Les annotations disponibles correspondent aux méthodes HTTP, à savoir : `@Get`, `@Put`, `@Delete`, `@Post` `@Options` et `@Patch`.

Ces annotations peuvent prendre un paramètre de type String pour indiquer les types MIME que la méthode peut renvoyer (cas de `Get` par exemple) ou recevoir (cas des `Put`). La valeur de la chaîne ne se résume qu'à la partie sous-type du format MIME (ex : au lieu de renseigner "application/json", on renseignera directement "json"). Si la méthode de traitement de la classe renvoie un objet java quelconque, le finder sous-jacent traduira automatiquement l'objet dans la représentation indiquée dans le paramètre de l'annotation (sous réserve qu'il existe une conversion possible). Par exemple, dans le cas de la représentation en JSON, la plateforme utilise directement l'utilitaire Jackson qui permet de traduire des objets java sous un format JSON et vice-versa. La liste des types MIME acceptés doivent s'écrire dans un ordre de préférence où chaque élément est séparé par une barre verticale. Par exemple, si une méthode est annotée `@Get(xml|json)`, le résultat sera par défaut en xml mais il peut être sous le format JSON si la requête a indiquer préférer recevoir du JSON. Si la méthode jette une exception alors une erreur 500 Internal Server Error sera renvoyée au client.

Tuto 2 – Programmation d'un mini-application Restlet

Dans ce tutoriel, nous allons montrer pas à pas comment construire une application Restlet qui maintient un annuaire d'étudiants.

Un étudiant est caractérisé par son numéro d'étudiant, son nom, son prénom et son numéro de téléphone.

Étape 1

Dans un premier temps, nous allons créer une classe `Etudiant` :

```
1 package srcs.restlet.annuaire;  
2  
3 public class Etudiant {  
4  
5     private String id;  
6     private String nom;  
7     private String prenom;  
8     private String telephone;  
9  
10    public Etudiant() {}  
11}
```

```

12 public Etudiant(String id, String nom, String prenom, String telephone) {
13     super();
14     this.id = id;
15     this.nom = nom;
16     this.prenom = prenom;
17     this.telephone = telephone;
18 }
19 public String getTelephone() {
20     return telephone;
21 }
22 public void setTelephone(String telephone) {
23     this.telephone = telephone;
24 }
25 public String getId() {
26     return id;
27 }
28 public String getNom() {
29     return nom;
30 }
31 public String getPrenom() {
32     return prenom;
33 }
34
35 @Override
36 public String toString() {
37     return "Etudiant [id=" + id + ", nom=" + nom + ", prenom=" + prenom + ",
38         telephone=" + telephone + "]";
39 }
40
41
42
43
}

```

Étape 2

Nous allons écrire une classe `Annuaire` qui sera une application Restlet. Cette classe est donc une sous-classe de `Application`. Elle contient une table associative qui associe l'identifiant d'un étudiant à une instance d'étudiant (ceci permet d'indexer les étudiants par leur numéro).

```

1 public Annuaire extends Application{
2     private Map<String,Etudiant> annuaire = new ConcurrentHashMap<>();
3

```

Étape 3

Nous ajoutons un constructeur par défaut qui permet de remplir la map au moment de linitialisation de lapplication.

```

1 public Annuaire() {
2     annuaire.put("45652", new Etudiant("45652", "Macron", "Emmanuel", "0956874123"));
3     annuaire.put("56423", new Etudiant("56423", "Seize", "Louis", "0864287951"));
4     annuaire.put("78951", new Etudiant("78951", "Chirac", "Jacques", "0864287951"));
5     annuaire.put("89546", new Etudiant("89546", "Sarkozy", "Nicolas", "0978654123"));
6

```

Étape 4

Nous allons redéfinir la méthode `createInboundRoot` qui sera appelée par la plateforme lors du déploiement de l'application. Cette méthode doit renvoyer le restlet racine de l'application. Ici ce sera un router.

```
1  @Override  
2  public Restlet createInboundRoot() {  
3      Router res = new Router();  
4  
5      return res;  
6  }
```

Étape 5

Nous allons définir une méthode statique `main` dans la classe `Annuaire`, permettant de construire le component et d'y associer une socket d'écoute pour le protocole HTTP. Nous utilisons l'hôte virtuel par défaut. Le path de l'URI d'accès à l'application est préfixé par "`/annuaire`".

```
1  public static void main(String[] args) throws Exception {  
2      Component c = new Component();  
3      c.getServers().add(Protocol.HTTP, 8585);  
4      c.getDefaultHost().attach("/annuaire", new Annuaire());  
5      c.start(); //démarrage des services  
6  }
```

Étape 6

Nous devons maintenant définir les ressources du serveur. La première ressource que nous allons exposer est l'ensemble des étudiants à savoir la table associative au complet. Pour cela, nous allons définir une classe `All` qui va hériter de la classe `ServerResource`. Afin de tout regrouper dans un seul fichier java, nous cette classe sera statique et interne à la classe `Annuaire`.

```
1  public static class All extends ServerResource {  
2      //voir étape suivante pour la suite  
3  }
```

Étape 7

Dans la classe `All`, nous allons définir une méthode `request()` (le nom n'a pas d'importance) et l'annoter avec `@Get`. La méthode peut renvoyer par ordre de préférence du XML et du JSON. L'annotation sera donc paramétrée par la chaîne `xml|json`

```
1  @Get("xml|json")  
2  public Map<String,Etudiant> request() {  
3      //voir étape suivante pour la suite  
4  }
```

Étape 8

Le corps de la méthode `request()` doit d'abord récupérer une référence sur l'application courante de la requête (accessible via la méthode `getApplication()` de la classe `ServerResource`) pour avoir accès à la map.

```
1  Application app = this.getApplication();  
2  if(! (app instanceof Annuaire)) {  
3      throw new ResourceException(Status.SERVER_ERROR_INTERNAL);  
4  }  
5  Annuaire a = (Annuaire) getApplication();  
6  return a.annuaire;
```

Étape 9

Une fois avoir démarré le serveur, on pourra remarquer qu'en lançant la commande suivante

```
curl http://localhost:8585/annuaire/etudiants
```

le retour sera une version XML de la map, alors que si on lance la commande suivante

```
curl -H "Accept: application/json" http://localhost:8585/annuaire/etudiants
```

le retour sera au format JSON

Étape 10

Un deuxième type de ressource que l'on peut renvoyer au client est un étudiant particulier en renseignant son identifiant dans le path de l'URI. Ainsi dans cet exemple, on pourrait obtenir les informations associées à l'étudiant d'identifiant *56423* avec l'URI

```
http://localhost:8585/annuaire/etudiants/56423
```

Pour ce faire il faut considérer des chemins à partie variable. Dans notre cas c'est la partie du chemin qui suffit l'URI. En pratique il faut déclarer une nouvelle route sur le routeur avec l'étiquette */etudiants/{id}*.

```
res.attach("/etudiants/{id}", EtudiantResource.class);
```

1

Étape 11

Nous allons maintenant définir la classe *EtudiantResource*. L'accès à la valeur de la variable se fait à l'aide de la méthode *getRequest().getAttributes().get("id")*;

```
public static class EtudiantResource extends ServerResource{
    @Get("xml|json")
    public Etudiant request() {
        Application app = this.getApplication();
        if(! (app instanceof Annuaire)) {
            throw new ResourceException(Status.SERVER_ERROR_INTERNAL);
        }
        Annuaire a = (Annuaire) getApplication();
        Object id = getRequest().getAttributes().get("id");
        if(!a.annuaire.containsKey(id)) {
            throw new ResourceException(Status.CLIENT_ERROR_NOT_FOUND);
        }
        return a.annuaire.get(id);
    }
}
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Étape 12

En testant avec la commande

```
curl http://localhost:8585/annuaire/etudiants/56423
```

on obtient la réponse

```
<Etudiant>
<id>56423</id>
<nom>Seize</nom>
<prenom>Louis</prenom>
<telephone>0864287951</telephone>
</Etudiant>
```

Tuto 3 – Client java et requêtes d’écriture

L’API de Restlet nous offre également des outils pour programmer des clients.

Étape 1

Une première chose à faire est d’instancier un objet de type `ClientResource` auquel on va passer en argument de son constructeur l’URI cible que l’on souhaite atteindre. Nous allons dans un premier temps télécharger les informations un étudiant particulier avec un `get` et l’afficher sur la console du client.

```
public class Client {  
    public static void main(String[] args) throws ResourceException, IOException {  
  
        ClientResource client = new ClientResource("http://localhost:8585/annuaire/  
            etudiants/56423");  
        client.get().write(System.out);  
    }  
}
```

1
2
3
4
5
6
7

La méthode `get` renvoie un objet de type `Representation` qu’il est possible d’envoyer dans un flux d’entrée/sortie grâce à la méthode `write`.

Étape 2

Pour transformer une représentation JSON en objet Java, il faut utiliser l’utilitaire `JackSon` ainsi :

```
client.accept(MediaType.APPLICATION_JSON);  
Representation r = client.get();  
JacksonRepresentation<Etudiant> jr = new JacksonRepresentation<>(r, Etudiant.class  
    );  
Etudiant e = jr.getObject();
```

1
2
3
4

À présent, nous allons exploiter les requêtes d’écriture en effaçant, ajoutant ou modifiant un étudiant.

Étape 3

Voici une extension du serveur pour offrir le service de supprimer un étudiant

```
@Delete  
public void supprimer() {  
    Application app = this.getApplication();  
    if(! (app instanceof Annuaire)) {  
        throw new ResourceException(Status.SERVER_ERROR_INTERNAL);  
    }  
    Annuaire a = (Annuaire) getApplication();  
    Object id = getRequest().getAttributes().get("id");  
    if(!a.annuaire.containsKey(id)) {  
        throw new ResourceException(Status.CLIENT_ERROR_NOT_FOUND);  
    }  
    a.annuaire.remove(id);  
}
```

1
2
3
4
5
6
7
8
9
10
11
12
13

Étape 4

Le code du client sera donc `client.delete()`. On pourra remarquer que la suppression a bien été appliquée si on refait un `get` sur la ressource qui doit renvoyer une erreur.

Étape 5

Pour envoyer un objet depuis un client pour créer ou modifier une ressource existante sur le serveur, il faut appeler la méthode `post` ou `put` en passant une représentation de l'objet en paramètre. La représentation que nous utiliserons ici est JSON.

```
Etudiant e = new Etudiant("56423", "Coty", "Rene", "089657484");  
client.post(new JacksonRepresentation<Etudiant>(e));
```

1
2

Étape 6

Côté serveur, il faut définir une méthode annotée par `@Post` et paramétriser cette annotation par `"json"` pour indiquer que c'est la seule représentation acceptée pour ce type d'opération.

```
@Post("json") //  
public void ajouter(Representation r) throws IOException {  
    JacksonRepresentation<Etudiant> jr = new JacksonRepresentation<>(r, Etudiant.  
        class);  
    Etudiant e = jr.getObject();  
    Application app = this.getApplication();  
    if(! (app instanceof Annuaire)) {  
        throw new ResourceException(Status.SERVER_ERROR_INTERNAL);  
    }  
    Annuaire a = (Annuaire) getApplication();  
    Object id = getRequest().getAttributes().get("id");  
    if(!id.equals(e.getId())) {  
        throw new ResourceException(Status.CLIENT_ERROR_CONFLICT);  
    }  
    a.annuaire.put(id.toString(), e);  
}
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15