Services Web REST

Support de cours version 3

2017-2018

Prof. El Habib NFAOUI

Université Sidi Mohamed Ben Abdellah de Fès

Faculté des Sciences Dhar El Mahraz Fès

Email: elhabib.nfaoui@usmba.ac.ma

Contenu

- Approche et spécifications REST
- Développement de services web REST à l'aide de JAX-RS
- Annexe : Protocole HTTP

APPROCHE ET SPÉCIFICATIONS REST

Plan

1. Présentation des services web REST
2. L'approche REST
3. Spécifications des services Web REST
4. REST vs SOAP

1. Présentation des services web REST

- On distingue actuellement deux familles de Services Web:
  - Les services Web SOAP (appelés aussi services Web étendus) qui se basent sur la pile (SOAP, WSDL, WS-*).
  - Les services Web REST (Representational State Transfer) qui permettent de construire une application pour les systèmes distribués comme le Web. Ce n'est pas un protocole ou un format, mais une architecture (celle de HTTP).

- REST (Representational State Transfer) 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 connaître le protocole HTTP (Hypertext Transfer Protocol), le principe des URI (Uniform Resource Identifier) et respecter quelques règles. **Il faut raisonner en termes de ressources.**

- Dans l'architecture REST, toute information est une ressource et chacune d'elles est désignée par une URI (Uniform Resource Identifier) – 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. Avec REST, les clients et les serveurs échangent des représentations des ressources en utilisant une interface et un protocole bien définis. Ces principes encouragent la simplicité, la légèreté et l'efficacité des applications. La figure ci-dessous résume ces principes.

1. Présentation des services web REST

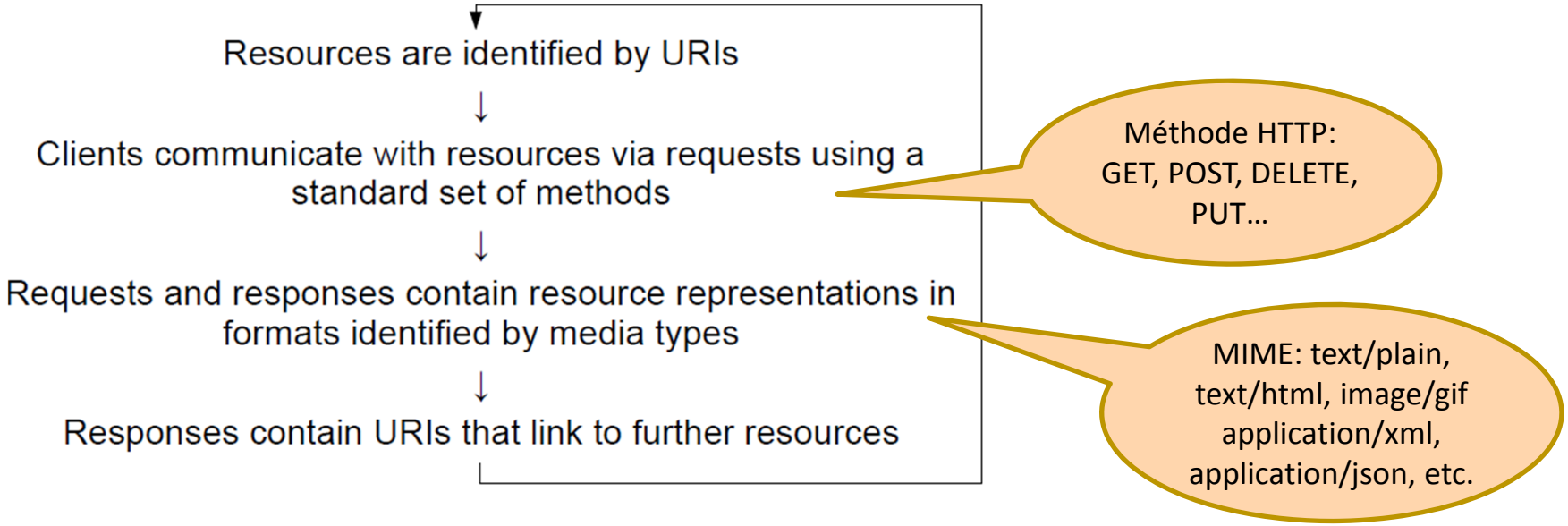


Figure 1. RESTful Application Cycle

1.1 Ressources et URI

- Les ressources jouent un rôle central dans les architectures REST. **Une ressource est tout ce que peut désigner ou manipuler un client, toute information pouvant être référencée dans un lien hypertexte**. Elle peut être stockée dans une base de données, un fichier, etc.

  Exemple de ressources:

  - une liste des livres JEE publiés par l'édition Toubkal,
  - les données météorologiques de Casa en 2011,
  - vos informations de contact,
  - les photos intéressantes postées sur Flickr le 5/2/2010.

- 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 (Universal Resource Identifiers) et, enfin, les combinaisons d'URL (Uniform Resource Locators) et d'URN (Uniform Resource Names). Voici quelques exemples d'URI :

  - http://www.toubkal.ma/catalogue/programmation/
  - http://www.biblio.ma/Resources/titles/Images/274L.gif
  - http://www.pearson.fr/contacts/
  - http://www.weather.com/weather/2008?location=Paris,France

1.2 Représentations

- On peut vouloir obtenir la représentation d'un objet sous forme de texte, de XML, de PDF ou sous un 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.

- Deux solutions sont possibles pour choisir entre les différentes représentations d'une ressource. La première consiste à proposer une URI par représentation. Cependant, en ce cas, les deux URI sont différentes et ne semblent pas directement liées. Voici un ensemble d'URI mieux organisé :
  - Représentation par défaut: http://www.apress.com/book/catalog/java
  - Représentation csv: http://www.apress.com/book/catalog/java.csv
  - Représentation xml: http://www.apress.com/book/catalog/java.xml

  L'autre solution consiste à n'exposer qu'une seule URI pour toutes les représentations (http://www.apress.com/book/catalog/java, par exemple) et à utiliser un mécanisme appelé négociation du contenu, que nous présenterons dans la section traitant le protocole HTTP.

1.3 HTTP, WADL

- HTTP, un protocole pour les systèmes d'informations distribués, collaboratifs et hypermédias, a conduit, avec les URI, HTML et les premiers navigateurs à la mise en place du World Wide Web. Le développement des services web REST exige une bonne maîtrise du fonctionnement de ce protocole. L'annexe 1 présente plus de détails sur ce protocole.

- Alors que les services web SOAP utilisent WSDL pour décrire le format des requêtes possibles, WADL (Web Application Description Language) sert à indiquer les interactions possibles avec un service web REST. Il est basé sur XML. Il facilite le développement des clients, qui peuvent ainsi charger et interagir directement avec les ressources. **WADL n'est pas obligatoire pour les services REST et il est peu utilisé.**

2. L’approche REST

- REST est un ensemble de contraintes de conceptions générales reposant sur HTTP. Dans cette section, nous étudions les concepts et les propriétés de REST.

2.1 Du Web aux services web

- 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.apress.com/books/. Le livre crée, la réponse renvoie l'URI de la nouvelle ressource : http://www.apress.com/books/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.apress.com/books/123456.
  - Utiliser PUT pour modifier la ressource située à l'URI http://www.apress.com/books/123456.
  - Utiliser DELETE pour supprimer la ressource à l'URI http://www.apress.com/books/123456.

- En se servant de verbes HTTP, nous pouvons donc effectuer toutes les actions CRUD (*Create, Read, Update, Delete)* sur une ressource.
- REST applique les mêmes principes de la pratique de la navigation sur le Web à vos services où à titre d'exemple des livres, des résultats des recherches, une table des matières ou une couverture d'un livre peuvent être définis comme des ressources.

2.2 Interface uniforme

- Le protocole de facto du Web est HTTP – un protocole standardisé de requête/réponse entre un client et un serveur –, et c'est l'interface uniforme des services web REST. Les services reposant sur SOAP, WSDL et les autres standards WS-* l'utilisent également au niveau de la couche transport, mais ils ne se servent que d'une infime partie de ses possibilités. S'il faut analyser le WSDL pour découvrir la sémantique d'un service SOAP puis appeler les bonnes méthodes, **les services web REST, en revanche, ont une interface uniforme (les verbes HTTP et les URI) : dès que l'on connait l'emplacement d'une ressource (son URI), il suffit d'invoquer les verbes HTTP (GET, POST, etc.)**.

  Outre l'avantage de sa familiarité, une interface web met en avant l'interopérabilité entre les applications : HTTP est largement répandu et le nombre de bibliothèques HTTP clientes garantit que nous n'aurons pas besoin de nous soucier des problèmes de communication.

2.3 Accessibilité et Connectivité

- Le second principe de la conception des services REST est l'accessibilité. Un service web doit rendre une application aussi accessible que possible, **ce qui signifie que chaque fragment d'information intéressante de cette application doit être une ressource et posséder une URI afin de pouvoir être aisément atteinte**. C'est la seule information qu'il faut publier pour rendre la ressource accessible. Les URI uniques permettent de créer des liens vers les ressources et, comme elles sont exposées *via une interface uniforme,* chacun sait exactement comment interagir avec elles : ceci permet d'utiliser une application de façon insoupçonnée.

- REST demande à ce que les ressources soient les plus connectées possible. C'est le fameux principe "L'hypermédia est le moteur de l'état de l'application". Ce principe provient, la encore, d'un examen du succès du Web. Les pages contiennent des liens pour passer de page en page de façon logique et simple, et l'on peut donc considérer que le Web est très bien connecté. S'il existe une relation forte entre deux ressources, celles-ci doivent être connectées. REST précise que les services web devraient également tirer parti de l'hypermédia pour informer le client de ce qui est disponible et de la façon de s'y rendre. Il met en avant la découverte des services. A partir d'une seule URI, un agent utilisateur accédant à un service bien connecté pourra découvrir toutes les actions et ressources disponibles, leurs différentes représentations, etc.

2.4 Sans état

- Une autre fonctionnalité de REST est l'absence d'état, ce qui signifie que toute requête HTTP est totalement indépendante puisque le serveur ne mémorisera jamais les requêtes qui ont été effectuées. Pour plus de clarté, l'état d'une ressource et celui de l'application sont généralement différenciés : l'état de la ressource doit se trouver sur le serveur et être partagé par tous, tandis que celui de l'application doit rester chez le client et être sa seule propriété. Par exemple, nous prenons le cas d'un client qui a cherché et a obtenu une représentation d'un livre. Alors l'état de l'application est que le client a récupéré une représentation du livre désiré, mais le serveur ne mémorisera pas cette information. L'état de la ressource, quant à lui, est l'information sur l'ouvrage : le serveur doit évidemment la mémoriser et le client peut le modifier. Si le panier virtuel est une ressource dont l'accès est réservé à un seul client, l'application doit stocker l'identifiant de ce panier dans la session du client.
- L'absence d'état a de nombreux avantages, notamment une meilleure adaptation à la charge: aucune information de session à gérer, pas besoin de router les requêtes suivantes vers le même serveur, gestion des erreurs, etc. Si vous devez mémoriser l'état, le client devra faire un travail supplémentaire pour le stocker.

3. Spécifications des services Web REST

- Contrairement à SOAP et à la pile WS-*, qui reposent sur les standards du W3C ; REST est un style d'architecture respectant certains critères de conception. Les applications REST, cependant, dépendent fortement d'autres standards :
  - HTTP ;
  - URI, URL ;
  - XML, JSON, HTML, GIF, JPEG, etc. (représentation des ressources).

- REST est comme un patron de conception : c'est une solution réutilisable d'un problème courant, qui peut être implémentée en différents langages.

3.1 Spécifications Java et implémentation de référence des services Web REST

Spécifications:

- La prise en compte par Java de l'architecture REST a été spécifiée par JAX-RS (Java API for RESTful Web Services). JAX-RS définit un ensemble d'API mettant en avant une architecture REST. Au moyen d'annotations, il simplifie l'implémentation de ces services et améliore la productivité.
- JAX-RS permet entre autres:
  - Le support de beans de session sans état comme ressources racine.
  - L'injection des ressources externes (gestionnaire de persistance, sources de données, EJB, etc.) dans une ressource REST.
  - Les annotations JAX-RS peuvent s'appliquer à l'interface locale d'un bean ou directement à un bean sans interface.

Implémentation de référence

- Jersey est l'implémentation de référence officielle de JAX-RS. Il existe également d'autres implémentations de JAX-RS, comme CXF (Apache), RESTEasy (JBoss)

4. REST vs SOAP

- **Des standards d'interopérabilité:**

Les deux familles de service web SOAP et REST facilitent l'usage des services par les consommateurs, notamment par un couplage technologique faible qui permet d'ignorer la technologie d'implémentation du service utilisé. Les solutions SOAP/WSDL (ou WS-*) sont défendues par les supporteurs des approches structurées et les solutions REST sont promues par les amateurs des solutions légères tant pour les développeurs que les consommateurs de services.

- **REST plus souvent choisi**

S'il est possible d'implémenter les expositions de services avec ces deux approches, les expositions externes privilégient la technologie REST pour la simplicité de la partie cliente, ce qui favorise les clients mobiles, plus contraints que les clients de type services d'entreprise ou applications d'entreprise. Si les besoins de fiabilité et de sécurité de message sont toujours d'actualité, il reste tout à fait possible d'utiliser des services SOAP/WSDL, éventuellement en sus.

DÉVELOPPEMENT DE SERVICES WEB REST À L'AIDE DE JAX-RS

Plan

1. Le modèle JAX-RS
2. Définition des URI
3. Extraction des paramètres
4. Consommation et production des types de contenus
5. Fournisseurs d'entités
6. Méthodes ou interface uniforme
7. Informations contextuelles
8. Gestion des exceptions
9. Cycle de vie
10. JAX-RS Client API

1. Le modèle JAX-RS

- JAX-RS est une API qui permet de décrire une ressource à l'aide de quelques annotations seulement.
- Les exigences que doit satisfaire une classe pour devenir un service REST sont les suivantes :
  - Elle doit être annotée par @javax.ws.rs.Path.
  - Pour ajouter les fonctionnalités des EJB au service REST, la classe doit être annotée par @javax.ejb.Stateless.
  - Elle doit être publique et ne pas être finale ni abstraite.
  - Les classes ressources racine (celles ayant une annotation @Path) doivent avoir **un constructeur par défaut public**. Les classes ressources non racine n'exigent pas ce constructeur.
  - La classe ne doit pas définir la méthode finalize().

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. JAX-RS étant indépendant du conteneur, les ressources peuvent être déployées dans un grand nombre de conteneurs de servlets.

1. Le modèle JAX-RS

- Une ressource est alors un POJO possédant au moins une méthode annotée par @javax.ws.rs.Path.

Exemple: Une ressource livre simple

```
@Path("/book")
public class BookResource {
        @GET
        @Produces("text/plain")
        public String getBookTitle() {
                return "Java avancé";
    }
}
```

- BookResource étant une classe Java annotée par @Path, la ressource sera hébergée à l'URI **/book**. La méthode getBookTitle() est annotée par @GET afin d'indiquer qu'elle traitera les requêtes HTTP GET et qu'elle produit du texte (le contenu est identifie par le type MIME text/plain). Pour accéder à la ressource, il suffit d'un client HTTP, un navigateur, par exemple, pouvant envoyer une requête GET vers l'URL http://www.myserver.com/book.

1. Le modèle JAX-RS

- Les services REST peuvent être des beans de session sans état. Dans ce cas, on ajoute l'annotation **@Stateless**. Par exemple, le code ci-dessous représente un service REST pouvant consommer et produire les représentations XML et JSON d'un livre.

```
@Path("books")
@Stateless
@Produces({"application/xml", "application/json"})
@Consumes({"application/xml", "application/json"})
public class BookResource {
    ….
    …
}
```

- Il est aussi possible d'utiliser JAX-RS en **mode autonome** (déployer directement l'application (JAR) via Java SE). L'implémentation Jersey nécessite l'ajout d'un serveur web en mode embarqué. Les usages de ce mode sont:
  - Fournir des services web à une application type client lourd.
  - Pour les tests unitaires, fournir des Mock de services web.

2. Définition des URI

- La valeur de l'annotation @Path est relative à un chemin d'URI. Lorsqu'elle est utilisée sur des classes, celles-ci sont considérées comme des *ressources racine*, car elles fournissent la racine de l'arborescence des ressources et l'accès aux sous-ressources.

  Exemple:

```
@Path("/customers")
public class CustomersResource {
        @GET
        public List getCustomers() {
        // ...
        }
}
```

- Vous pouvez également intégrer dans la syntaxe de l'URI des templates de chemins d'URI : ces variables seront évaluées à l'exécution. La documentation Javadoc de l'annotation @Path décrit la syntaxe des templates :

```
@Path("/customers/{customername}")
```

- @Path peut également s'appliquer aux méthodes des ressources racine, ce qui permet de regrouper les fonctionnalités communes à plusieurs ressources, comme le montre le code ci-dessous.

Exemple: Une ressource article avec plusieurs annotations @Path

```
@Path("/items")
public class ItemsResource {
        @GET
        public List<Item> getListOfItems() {
            // ...
        }

        @GET
        @Path("{itemid}")
        public Item getItem(@PathParam("itemid") String itemid) {
            // ...
        }

        @PUT
        @Path("{itemid}")
        public void putItem(@PathParam("itemid") String itemid,Item item) {
            // ...
      }
```

Exemple (suite)

```
    @DELETE
    @Path("{itemid}")
    public void deleteItem(@PathParam("itemid") String itemid) {
        // ...
    }

    @GET
    @Path("/books/")
    public List<Book> getListOfBooks() {
        // ...
    }

    @GET
    @Path("/books/{bookid}")
    public Book getBook(@PathParam("bookid") String bookid) {
        // ...
    }
}
```

2. Définition des URI

- Si @Path est appliquée à la fois sur la classe et une méthode, le chemin relatif de la ressource produite par cette méthode est la concaténation de la classe et de la méthode. Pour obtenir un livre par son identifiant, par exemple, le chemin sera /items/books/1234. Si l'on demande la ressource racine /items, seule la méthode sans annotation @Path sera sélectionnée (getListOfItems(), ici). Si l'on demande /items/books, c'est la méthode getListOfBooks() qui sera invoquée. Si @Path("/items") n'annotait que la classe et aucune méthode, le chemin d'accès de toutes les méthodes serait le même et il faudrait utiliser le verbe HTTP (GET, PUT) et la négociation du contenu (texte, XML, etc.) pour les différencier.

3. Extraction des paramètres

- Nous avons besoin d'extraire des informations sur les URI et les requêtes lorsqu'on les manipule. JAX-RS fournit tout un ensemble d'annotations pour extraire les différents **paramètres qu'une requête** peut envoyer (@QueryParam, @MatrixParam, @CookieParam, @HeaderParam et @FormParam).

3.1 @PathParam

- L'annotation @PathParam permet d'extraire la valeur d'un paramètre template d'une URI. Le code suivant, par exemple, extraira l'identifiant du client (98342) de l'URI http://www.myserver. com/customers/98342 :

```
@Path("/customers")
public class CustomersResource {
  @GET
  @Path("{customerid}")
  public Customer getCustomer(@PathParam("customerid") Long customerid) {
            // ...
        }
}
```

3.2 @QueryParam

- L'annotation @QueryParam extrait la valeur du paramètre d'une requête. Le code suivant extraira le code postal (19870) de l'URI

  http://www.myserver.com/customers?zip=19870 :

```
@Path("/customers")
public class CustomersResource {
        @GET
        public Customer getCustomerByZipCode(@QueryParam("zip") Long zip) {
            // ...
        }
}
```

3.3 @MatrixParam

- L'annotation @MatrixParam agit comme @QueryParam, sauf qu'elle extrait la valeur d'un paramètre matrice d'une URI (le délimiteur est **;** au lieu de ?). Elle permet, par exemple, d'extraire le nom de l'auteur (XXXX) de cette URI :

  http://www.myserver.com/products/books;author=XXXX.

3.4 @CookieParam, @HeaderParam et @FormParam

- @CookieParam extrait la valeur d'un cookie, tandis que @HeaderParam permet d'obtenir la valeur d'un entête. Le code suivant, par exemple, extrait l'identifiant de session du cookie :

```
@Path("/products")
public class ItemsResource {
        @GET
        public Book getBook(@CookieParam("sessionid") int sessionid) {
            // ...
        }
}
```

- L'annotation @FormParam précise que la valeur d'un paramètre doit être extraite d'un formulaire situé dans le corps de la requête.

3.5 @DefaultValue

- On peut ajouter @DefaultValue à toutes les annotations définies avant pour définir une valeur par défaut pour le paramètre que l'on attend. Cette valeur sera utilisée si les métadonnées correspondantes sont absentes de la requête. Si le paramètre *age* ne se trouve pas dans la requête, par exemple, le code suivant utilisera la valeur 50 par défaut :

```
@Path("/customers")
public class CustomersResource {
        @GET
        public Response getCustomers(@DefaultValue("50") @QueryParam("age") int age) {
            // ...
        }
}
```

4. Consommation et production des types de contenus

- Avec REST, une ressource peut avoir plusieurs représentations : un livre peut être représenté comme une page web, un document XML ou une image affichant sa couverture. Les annotations @javax.ws.rs.Consumes et @javax.ws.rs.Produces peuvent s'appliquer à une ressource qui propose plusieurs représentations : elle définit les types de médias échangés entre le client et le serveur. Autrement dit, l'annotation @Consumes est utilisée pour spécifier le ou les types MIME qu'une méthode d'une ressource peut accepter. L'annotation @Produces est utilisée pour spécifier le ou les types MIME qu'une méthode d'une ressource peut produire.

  L'utilisation de l'une de ces annotations sur une méthode ressource redéfinit celle qui s'appliquait sur la classe de la ressource pour un paramètre d'une méthode ou une valeur de retour. En leur absence, on suppose que la ressource supporte tous les types de médias (*/*).
- La liste des constantes des différents types MIME est disponible dans la calsse MediaType.

Exemple 1

- Dans le code ci-dessous CustomerResource produit par défaut une représentation en texte brut, sauf pour certaines méthodes.

```
@Path("/customers")
@Produces("text/plain")
public class CustomersResource {
        @GET
        public String getAsPlainText() {
            // ...
        }
        @GET
        @Produces("text/html")
        public String getAsHtml() {
            // ...
        }
```

Exemple 1 (suite)

```
    @GET
    @Produces("application/json")
    public List<Customer> getAsJson() {
        // ...
    }

    @PUT
    @Consumes("text/plain")
    public void putBasic(String customer) {
        // ...
    }

    @POST
    @Consumes("application/xml")
    public Response createCustomer(InputStream is) {
        // ...
    }
}
```

4. Consommation et production des types de contenus

- Si une ressource peut produire plusieurs types de médias Internet, la méthode choisie correspondra au type qui convient le mieux à l'en-tête Accept de la requête HTTP du client. Si, par exemple, cet en-tête est :

  Accept: text/plain

  et que l'URI est /customers, c'est la méthode getAsPlainText() qui sera invoquée.

  Le client aurait également pu utiliser l'en-tête suivant :

  Accept: text/plain; q=0.8, text/html

  Cet en-tête annonce que le client peut accepter les types text/plain et text/html mais qu'il préfère le dernier avec un facteur de qualité de 0.8 ("je préfère huit fois plus le text/html que le text/plain"). En incluant cet en-tête dans une requête adressée à /customers, c'est la méthode getAsHtml() qui sera invoquée.

Exemple 2 : Représentation JSON d'une liste de clients

- Dans le code ci-dessous, la méthode getAsJson() renvoie une représentation de type application/json. JSON est un format léger pour l'échange de données : comme vous pourrez le constater, il est moins verbeux et plus lisible que XML. Jersey convertira pour vous la liste des clients au format JSON en utilisant les fournisseurs d'entités prédéfinis MessageBodyReader et MessageBodyWriter.

```
@GET
@Produces("application/json")
public List<Customer> getAsJson() {
    // ...
}
```

```
{"customers": {
    "id": "0",
    "version": "1",
    "customer": [
      {
        "firstName": "Said",
        "lastName": "MIDON",
        "address": {
          "streetAddress": "21 2nd Street",
          "city": "Casablanca",
          "postalCode": 20000
        },
        "phoneNumbers": [
          "212 555-1234",
          "646 555-4567"
        ]
      },
```

```
{
        "firstName": "Hassan",
        "lastName": "HASSANI",
        "address": {
          "streetAddress": "2 Rue des ERRAZI",
          "city": "Fès",
          "postalCode": 30000
        },
        "phoneNumbers": [
          "+212 555-1234",
        ]
      }
    ]
  }
}

]
```


Info:

Jersey ajoute une association BadgerFish de JAXB XML vers JSON. BadgerFish (http://badgerfish.ning.com/) est une convention de traduction d'un document XML en objet JSON afin qu'il soit plus facile à manipuler avec JavaScript.

5. Fournisseurs d'entités

- Lorsque les entités (contenu du corps d'une requête ou d'une réponse) sont reçues dans des requêtes ou envoyées dans des réponses, l'implémentation JAX-RS doit pouvoir convertir les représentations en Java et vice versa : c'est le **rôle des fournisseurs d'entités**.

- Lorsqu'une méthode ressource est invoquée, **les paramètres annotés par l'une des annotations d'extraction vues précédemment sont initialisés.** La valeur d'un paramètre **non annoté** (appelé **paramètre entité**) est obtenue **à partir du corps de la requête** et convertie **par un fournisseur d'entités.**

- L'implémentation de JAX-RS offre un ensemble de fournisseurs d'entités par défaut convenant aux situations courantes (voir Tableau ci-dessous). Ils permettent d'effectuer automatiquement des opérations de sérialisation et de désérialisation vers un type Java spécifique.

Fournisseurs par défaut de l'implémentation de JAX-RS

Type	*Description*
`byte[]`	Tous types de médias (`*/*`)
`java.lang.String`	Tous types de médias (`*/*`)
`java.io.InputStream`	Tous types de médias (`*/*`)
`java.io.Reader`	Tous types de médias (`*/*`)
`java.io.File`	Tous types de médias (`*/*`)
`javax.activation.DataSource`	Tous types de médias (`*/*`)
`javax.xml.transform.Source`	Types XML (`text/xml`, `application/xml` et `application/-*+xml`)
`javax.xml.bind.JAXBElement`	Types de médias XML de JAXB (`text/-xml`, `application/ xml` et `application/*+xml`)
`MultivaluedMap<String,String>`	Contenu de formulaire (`application/x-www-form-urlencoded`)
`javax.ws.rs.core StreamingOutput`	Tous types de médias (`*/*`), uniquement `MessageBodyWriter`

5. 1 Exemples

Nous montrons ici quelques exemples d'utilisation des fournisseurs par défaut de l'implémentation JAX-RS.

Exemple 1: requête et réponse avec un flux d'entrée (InputStream)

```
@Path("/customer")
public class CustomerResource {
	@PUT
	@Path("inputstream")
	public void updateCustomersWithInputStream(InputStream is)  {
		//traiter l'objet is de type InputStream
	}

	@Path("inputstream")
	@GET
	@Produces(MediaType.TEXT_XML)
	public InputStream getCustomersWithInputStream() {
		return new FileInputStream("d:/exemple.xml");
	}
}
```

Exemple 2: requête et réponse avec un fichier (File)

```
@Path("/customer")
public class CustomerResource {
        @Path("file")
        @PUT
        public void updateCustomerWithFile(File file) {
                //traiter l'objet file de type File
        }
        @Path("file")
        @GET
        @Produces(MediaType.TEXT_XML)
        public File getCustomersWithFile() {
                File file = new File("c:\\example.xml");
                return file;
        }
        ...
}
```

Exemple 3: requête et réponse avec un String

```
@Path("/customer")
public class CustomerResource {
      @Path("string")
      @PUT
      public void updateCustomerWithString(String s) {
            //traiter l'objet s de type String
      }
      @Path("string")
      @GET
      @Produces(MediaType.TEXT_XML)
      public String getCustomerWithString() {
            ……
            return "<?xml version=\"1.0\"?>" +
                  "<customer>"+
                  "<lastName> NFAOUI </lastName>"+
                  "<address> Fes </address>"+
                  "</customer>";
      }
      …
}
```

Exemple 4: requête et réponse avec un type personnalisé

JAX-RS permet d'utiliser directement des types (classes) personnalisés en s'appuyant sur la spécification JAXB (Java Architecture for XML Binding).

La classe Customer ci-dessous est annotée avec @XmlRootElement pour définir l'élément racine de l'arbre XML.

```
@XmlRootElement(name = "customer")
public class Customer {
    private String firstName;
    private String lastName;
    …
    //Constructors, getters and setters
}
```

Exemple 4: requête et réponse avec un type personnalisé

```
@Path("/customer")
public class CustomerResource {
        @Consumes("application/xml")
        @PUT
        public void updateCustomerWithJAXBXML(Customer customer) {
                //traiter l'objet customer
        }

        @GET
        @Produces("application/xml")
        public Customer getCustomerWithJAXBXML() {
                ....
                return new Customer(…);
        }
        ...
}
```

Exemple 4: requête et réponse avec un type personnalisé

L'exemple ci-dessous montre l'utilisation d'un objet **JAXBElement** pour envelopper le type Customer.

```
@Path("/customers")
public class CustomersResource {
        @Context
        UriInfo uriInfo
        ...

         @POST
         @Consumes({"application/xml","application/json"})
         public Response createCustomer(JAXBElement<Customer> customerJaxb) {
                Customer customer= customerJaxb.getValue();
                …….
                URI customerUri = uriInfo.getAbsolutePathBuilder().path(customer.getId().toString()).build();
                return Response.created(customerUri).build();
         }
}
```

5.2 Création de fournisseurs d'entités personnalisés

- Il existe deux variantes de fournisseurs d'entités : MessageBodyReader et MessageBodyWriter. Pour traduire le corps d'une requête en Java, une classe doit implémenter l'interface **javax.ws.rs.ext.MessageBodyReader** et être annotée par **@Provider**. Par défaut, l'implémentation est censée consommer tous les types de médias (*/*), mais l'annotation @Consumes permet de restreindre les types supportés. De la même façon, un type Java peut être traduit en corps de réponse. Une classe Java voulant effectuer ce traitement doit implémenter l'interface **javax.ws.rs.ext.MessageBodyWriter** et être annotée par l'interface **@Provider**. L'annotation @Produces indique les types de médias supportés.

5.2 Création de fournisseurs d'entités personnalisés

Exemple:

Le code ci-dessous montre un fournisseur convertissant une entité Customer en un corps HTTP en texte brut.

```
@Provider
@Produces("text/plain")
public class CustomerWriter implements MessageBodyWriter<Customer> {
    public boolean isWriteable(Class<?> type) {
        return Customer.class.isAssignableFrom(type);
    }
    public void writeTo(Customer customer, MediaType mediaType,
                        MultivaluedMap<String, Object> headers, OutputStream outputStream)
                                                                    throws IOException {
        outputStream.write(customer.get(id).getBytes());
        // ...
    }
}
```

6. Méthodes ou interface uniforme

- Nous avons vu comment le protocole HTTP gérait ses requêtes, ses réponses et ses méthodes d'actions (GET, POST, PUT, etc.). JAX-RS définit ces méthodes HTTP à l'aide d'annotations : @GET, @POST, @PUT, @DELETE, @HEAD et @OPTIONS. Seules les **méthodes publiques** peuvent être exposées comme des méthodes de ressources. Le code ci-dessous, par exemple, montre une ressource customer exposant les méthodes CRUD.

6. Méthodes ou interface uniforme

Exemple : Ressource customer exposant les opérations CRUD

```
@Path("/customers")
public class CustomersResource {

    @GET
    public List<Customer> getListOfCustomers() {
        // ...
    }

    @POST
    @Consumes("application/xml")
    public Response createCustomer(InputStream is) {
        // ...
    }
```

6. Méthodes ou interface uniforme

```
    @PUT
    @Path("{customerid}")
    @Consumes("application/xml")
    public Response updateCustomer(@PathParam("customerid") String customerId, InputStream is)
        {
        // ...
    }

    @DELETE
    @Path("{customerid}")
    public void deleteCustomer(@PathParam("customerid") String customerId) {
        // ...
    }
}
```

6. Méthodes ou interface uniforme

- Lorsqu'une méthode ressource est invoquée, **les paramètres annotés par l'une des annotations d'extraction vues précédemment sont initialisés.** La valeur d'un paramètre **non annoté** (appelé **paramètre entité**) est obtenue **à partir du corps de la requête** et convertie **par un fournisseur d'entités.**

- Les méthodes peuvent renvoyer **void, Response ou un autre type Java**. La classe **Response** permet de définir des réponses complexes permettant de : choisir un code d'état de retour, de retourner un URI, de fournir des paramètres dans l'entête, etc.

  A titre d'exemple, lorsque l'on crée un nouveau client (Customer) avec la méthode createCustomer() de l'exemple précédent, il faudra renvoyer son URI.

6.1 Classe Reponse

La classe **Response** dispose de méthodes abstraites non utilisables directement.

Quelques méthodes (se reporter à Java Doc pour plus de détails):

<<abstract>> Response
+ <<abstract>> getEntity():Object + <<abstract>> getStatus():int + created(location: URI): Response.ResponseBuilder + ok(): Response.ResponseBuilder + serverError(): Response.ResponseBuilder + status(status:int):Response.ResponseBuilder

JavaDoc: Method Summary

- public abstract Object getEntity(): Get the message entity Java instance. Returns null if the message does not contain an entity body.
- public abstract int getStatus(): Get the status code associated with the response.
- public static Response.ResponseBuilder status(int status): Create a new ResponseBuilder with the supplied status.

6.1 Classe Reponse

Quelques méthodes de la classe **Response.ResponseBuilder** (se reporter à Java Doc pour plus de détails):

<<static>> Response.ResponseBuilder
+ <<abstract>> build():Response + <<abstract>> entity(entity:Object):ResponseBuilder + header(name:String, value: Object):ResponseBuilder

6.2 Classe Response: Exemple

Exemple 1: la méthode getInfo() retourne une réponse contenant un code de statut, un paramètre d'entête et un corps contenant une chaîne de caractères. La méthode createCustomer() retourne une réponse contenant un URI.

```
@Path("/customers")
public class CustomersResource {
        @Context
        UriInfo uriInfo
        ...
        @GET
        public Response getInfo() {
                .....
                 return Response
                 .status(Response.Status.OK)
                 .header("param1", "valeur")
                 .entity("Corps de la réponse")
                 .build();
        }
        @POST
        @Consumes({"application/xml","application/json"})
        public Response createCustomer(JAXBElement<Customer> customerJaxb) {
              Customer customer= customerJaxb.getValue();
              URI customerUri = uriInfo.getAbsolutePathBuilder().path(customer.getId().toString()).build();
              return Response.created(customerUri).build();
        }
}
```

7. Informations contextuelles

- Le fournisseur de ressources a besoin d'informations contextuelles pour traiter correctement une requête. L'annotation **@javax.ws.rs.core.Context** sert à injecter les classes suivantes dans un **attribut** ou dans le **paramètre d'une méthode** : HttpHeaders (informations liées à l'entête), UriInfo (informations liées aux URIs), Request (informations liées au traitement de la requête), SecurityContext (informations liées à la sécurité) et Providers.

- Certains de ces objets permettent d'obtenir les mêmes informations que les précédentes annotations liées aux paramètres.

Exemple : Ressource customer obtenant une UriInfo

Exemple:

Le code de l'exemple ci-dessous reçoit par injection une UriInfo afin de pouvoir construire des URI.

```
@Path("/customers")
public class CustomersResource {
        @Context
        UriInfo uriInfo;

        @GET
        @Produces("application/json")
        public JSONArray getListOfCustomers() {
                JSONArray uriArray = new JSONArray();
                for (Customer customer : customerDAO.findAll()) {
                        UriBuilder ub = uriInfo.getAbsolutePathBuilder();
                        URI userUri = ub.path(customer.getCustomerId()).build();
                        uriArray.put(userUri.toASCIIString());
                }
                return uriArray;
        }
}
```

7.1 En-têtes

- Un objet de type **javax.ws.rs.core.HttpHeaders** permet aux développeurs de ressources d'extraire les informations contenues dans l'entête d'une requête. Une instance de **HttpHeaders** peut être injectée dans un **attribut ou dans un paramètre de méthode** au moyen de l'annotation **@Context** afin de permettre d'accéder aux valeurs des en-têtes sans tenir compte de leur casse.
- Quelques méthodes:

<<interface>> **HttpHeaders**
+ getCookies(): Map<String, Cookie> + getLanguage(): Locale + getRequestHeaders(): MultivaluedMap<String,String> + getMediaType(): MediaType

JavaDoc: Method Summary

- getCookies(): Get any cookies that accompanied the request.
- getLanguage(): Get the language of the request entity.
- getRequestHeaders(): Get the values of HTTP request headers.
- getMediaType(): Get the media type of the request entity.

Exemple

Si le service fournit des ressources localisées, par exemple, le code suivant permettra d'extraire la valeur de l'en-tête **Accept-Language** :

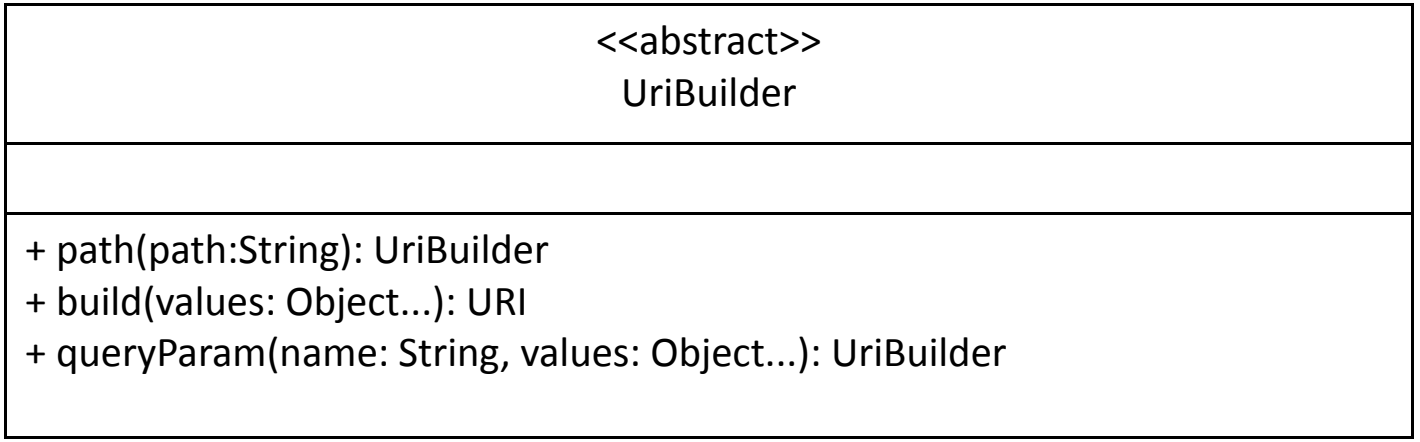
```
@GET
@Produces{"text/plain"}
public String get(@Context HttpHeaders headers) {
    List<java.util.Locale> locales =headers.getAcceptableLanguages();
    // ou headers.getRequestHeader("accept-language");
    ...
}
```

7.2 Construction d'URI

- Les liens hypertextes sont un élément central des applications REST. Pour évoluer à travers les états de l'application, les services web REST doivent gérer les transitions et la construction des URI. Pour ce faire, JAX-RS fournit une classe **javax.ws.rs.core.UriBuilder** destinée à remplacer java.net.URI et à faciliter la construction d'URI. UriBuilder dispose d'un ensemble de méthodes permettant de construire de nouvelles URI ou d'en fabriquer à partir d'URI existantes.

7.2 Construction d'URI

- Quelques méthodes de UriBuilder (se reporter à Java Doc pour plus de détails):



JavaDoc: Method Summary

- path(path:String): Append path to the existing path.
- build(Object... values): Build a URI, using the supplied values in order to replace any URI template parameters.

Exemple: Méthode renvoyant un tableau d'URI

Exemple: Le code ci-dessous se sert d'un objet UriBuilder pour renvoyer un tableau d'URI.

```
@Path("/customers/")
public class CustomersResource {
        @Context UriInfo uriInfo;

        @GET
        @Produces("application/json")
        public JSONArray getCustomersAsJsonArray() {
                JSONArray uriArray = new JSONArray();
                for (UserEntity userEntity : getUsers()) {
                UriBuilder ub = uriInfo.getAbsolutePathBuilder();
                URI userUri = ub.path(userEntity.getUserid()).build();
                uriArray.put(userUri.toASCIIString());
            }
         return uriArray;
        }
}
```

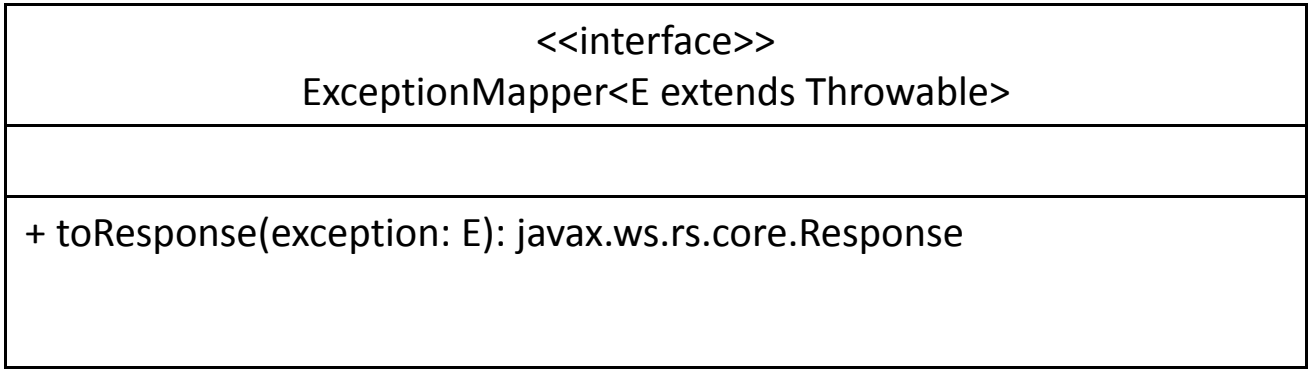
8. Gestion des exceptions

- Comme le montre le code ci-dessous, nous pouvons lever à tout instant une exception **WebApplicationException** ou l'une de ses sous-classes dans un fournissseur de ressources. Cette exception sera **capturée** par **l'implémentation de JAX-RS et convertie en réponse HTTP**. L'erreur par défaut est un code 500 avec un message vide, mais la classe javax.ws.rs.WebApplicationException offre différents constructeurs permettant de choisir un code d'état spécifique (défini dans l'énumération javax.ws.rs.core.Response.Status) ou une entité.

```
@Path("customers/{id}")
public Customer getCustomer(@PathParam("id") int id) {
        if(id < 1)
          throw new WebApplicationException(Response.status(400).entity("Id must be a
                        positive integer!"));
        Item i = em.find(Item.class, id);
        if (i == null)
          throw new WebApplicationException(Response.Status.NOT_FOUND);
        return i;
}
```

8. Gestion des exceptions

- JAX-RS offre la possibilité de créer des fournisseurs de traduction d'exception, qui traduisent une exception en un objet Response. Si cette exception est lancée, l'implémentation JAX-RS la capturera et appellera le fournisseur de traduction d'exception approprié. Ces fournisseurs sont des implémentations de l'interface ExceptionMapper<E extends java.lang.Throwable> annotées par @Provider.



JavaDoc: Method Summary

- Response toResponse(E exception) : Map an exception to a **Response**. Returning null results in a **Response.Status.NO_CONTENT** response. Throwing a runtime exception results in a **Response.Status.INTERNAL_SERVER_ERROR** response.

8. Gestion des exceptions

Exemple:

```
@Provider
public class EntityNotFoundMapper implements
                        ExceptionMapper<javax.persistence.EntityNotFoundException> {

    public Response toResponse( javax.persistence.EntityNotFoundException ex) {
        return Response.status(404).entity(ex.getMessage()).type("text/plain").build();
    }
}
```

9. Cycle de vie

- Lorsqu'une requête arrive, la ressource cible est résolue et une nouvelle instance de la classe ressource racine correspondante est créée. **Le cycle de vie d'une classe ressource racine dure donc le temps d'une requête**. Premièrement, le constructeur est appelé, ensuite les références de ressources sont injectées (injection de dépendances), puis la méthode appelée est exécutée et enfin l'objet devient candidat au ramasse miettes (garbage collector).

  S'ils sont déployés dans un conteneur Java EE (servlet ou EJB), les classes ressources et les fournisseurs JAX-RS peuvent également utiliser les annotations de gestion du cycle de vie et de la sécurité définies: @PostConstruct, @PreDestroy, @RunAs, @RolesAllowed, @PermitAll, @DenyAll et @DeclareRoles. Le cycle de vie d'une ressource peut se servir de @PostConstruct et de @PreDestroy pour ajouter de la logique métier respectivement après la création d'une ressource et avant sa suppression.

10. JAX-RS Client API

- JAX-RS Client est une API de haut niveau permettant d'accéder aux ressource REST. Elle est définie dans le package **javax.ws.rs.client**.
- Voici les étapes de base nécessaires pour accéder à une ressource REST en utilisant JAX-RS Client API:
  - Obtenir une instance de **javax.ws.rs.client.Client** interface.
  - Configurer cette instance avec un "target" basé sur un URI.
  - Créer une requête basée sur ce "target".
  - Demander la requête.

Exemple:

Le code ci-dessous permet d'invoquer le service REST en utilisant une requête HTTP de type GET, le type String est défini pour l'entité retournée.

```
Client client = ClientBuilder.newClient();
String name = client.target("http://example.com/webapi/hello")
                    .request(MediaType.TEXT_PLAIN)
                    .get(String.class);
```

ANNEXE : PROTOCOLE HTTP

Plan

1. Introduction
2. Requêtes et réponses
3. Méthodes de HTTP
4. Négociation du contenu
5. Types de contenu
6. Codes d'état
7. Mise en cache et requêtes conditionnelles

1. Introduction

- HTTP, un protocole pour les systèmes d'informations distribués, collaboratifs et hypermédias, a conduit, avec les URI, HTML et les premiers navigateurs à la mise en place du World Wide Web. Géré par le W3C (World Wide Web Consortium) et l'IETF (Internet Engineering Task Force), HTTP est le résultat de plusieurs RFC (Requests For Comment), notamment la RFC 2616, qui définit http 1.1. Il repose sur des requêtes et des réponses échangées entre un client et un serveur.

- Fonctionnement (très simple en HTTP/1.0)
  - Connexion
  - Envoi d'une requête HTTP, par exemple demande d'un document (méthode GET)
  - Réception d'une réponse HTTP
  - déconnexion

2. Requêtes et réponses

- Un client envoie une requête à un serveur afin d'obtenir une réponse (voir figure ci-dessous). Les messages utilisés pour ces échanges sont formés d'une enveloppe et d'un corps également appelé document ou représentation.

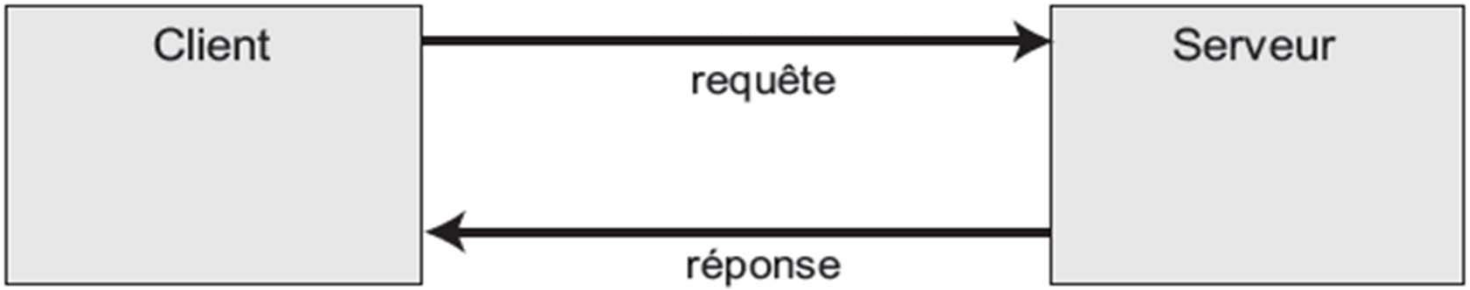


Figure 1. Requête et réponse http

2. Requêtes et réponses

- Voici, par exemple, un type de requête envoyée à un serveur en utilisant cURL (http://curl.haxx.se/) comme client web (client HTTP):

```
D:\>curl http://www.google.co.ma/index.html  -v
* About to connect() to www.google.co.ma port 80 (#0)
*   Trying 173.194.45.95...
* connected
* Connected to www.google.co.ma (173.194.45.95) port 80 (#0)
> GET /index.html HTTP/1.1
> User-Agent: curl/7.28.0
> Host: www.google.co.ma
```

- Cette requête contient plusieurs informations envoyées par le client :
  - la méthode de HTTP : GET ;
  - le chemin : /index.html ;
  - Le protocole et sa version: HTTP/1.1;
  - plusieurs autres en-têtes de requête comme User-Agent et Host.

Vous remarquez que la requête n'a pas de corps (un GET n'a jamais de corps).

2. Requêtes et réponses

- **Remarque:**

  Nous utilisons ici cURL (http://curl.haxx.se/) comme client, mais nous aurions pu également utiliser Firefox ou n'importe quel autre client web. CURL est un outil en ligne de commande permettant de transférer des fichiers par HTTP, FTP, Gopher, SFTP, FTPS, SCP, TFTP et bien d'autres protocoles encore en utilisant une URL. Avec lui, vous pouvez envoyer des commandes HTTP, modifier les en-têtes HTTP, etc. : c'est donc un outil parfait pour simuler les actions d'un utilisateur dans un navigateur web.

2. Requêtes et réponses

- En réponse, le serveur renverra le message suivant :

```
< HTTP/1.1 200 OK
< Date: Sun, 04 Oct 2015 07:35:25 GMT
< Content-Type: text/html; charset=ISO-8859-1
< Server: gws
< ……
<
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="fr-MA">
<head> ……..
```

- Cette réponse est formée des parties suivantes :
  - **Un code de réponse.** Ici, ce code est 200 OK.
  - **Plusieurs en-têtes de réponse.** Notamment Date, Server, Content-Type. Ici, le type du contenu est text/html, mais il pourrait s'agir de n'importe quel format comme du XML (application/xml) ou une image (image/jpeg).
  - **Un corps ou représentation.** Ici, il s'agit du contenu de la page web renvoyée (dont nous n'avons reproduit qu'un fragment).

2. Requêtes et réponses

La tableau ci-dessous présente quelques en-têtes HTTP

En-tête	Usage
Date	Date et heure de formulation de la requête.
Host	Domaine interrogé (permet d'héberger plusieurs noms de domaine sur le même serveur avec la même adresse IP).
Server	Nom du serveur répondant.
User-Agent	Chaîne de texte identifiant l'agent utilisateur (généralement nom du moteur de rendu du navigateur suivi du numéro de version et du système d'exploitation).
Content-Length	Longueur des données envoyées ou reçues dans la partie utile suivant les en-têtes (en octets).
Content-Type	Type MIME de la ressource envoyée par le serveur.
Expires	Date d'expiration de la ressource.
Last-Modified	Date de dernière modification de la ressource.
Accept	Types MIME acceptés.
Accept-Charset	Encodages de caractères acceptés.
Accept-Language	Langages de contenu acceptés.
Cache-Control	Directives de contrôle sur le cache.

3. Méthodes de 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

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

À 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 commentaire à un blog, 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).

3. Méthodes de HTTP

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 l'on envoie 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.

Autres Méthodes

Comme on l'a évoqué, il existe d'autres actions HTTP, plus rarement employées : HEAD, TRACE, OPTIONS, CONNECT.

4. Négociation du contenu

- La négociation du 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 : par exemple, la meilleure représentation pour l'utilisateur d'un téléphone portable peut, en effet, ne pas être la plus adaptée à un lecteur de flux RSS.

  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 HTML publiés par Apress, l'application cliente (l'agent utilisateur) demandera http://www.apress.com/book/catalog/html 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.

5. Types de contenu

- HTTP utilise des types de supports Internet (initialement appelés types MIME) dans les en-têtes Content-Type et Accept afin de permettre un typage des données et une négociation du 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/xml, 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. La définition complète dy type de support text/html se trouve dans la RFC 2854.
  - **text/plain**. Il s'agit du type de contenu par défaut car il est utilisé pour les messages textuels simples.
  - **image/gif, image/jpeg, image/png**. Le type de support image exige la présence d'un dispositif d'affichage (un écran ou une imprimante graphique, par exemple) permettant de visualiser l'information.
  - **text/xml, application/xml**. XML 1.0 a été publié par le W3C en février 1998. La définition complète de ce support est décrite dans la RFC 3023.
  - **application/json**. JSON (JavaScript Object Notation) est un format textuel léger pour l'échange de données. Il est indépendant des langages de programmation.

6. Codes 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 intégré 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.

6. Codes d'état

La tableau ci-dessous montre quelques codes d'état HTTP:

Code	Signification	Explication
200	OK	Succès.
301	Moved Permanently	Ressource déplacée de façon permanente à une autre adresse.
302	Found	Ressource déplacée de façon temporaire à une autre adresse.
304	Not modified	Ressource non modifiée depuis la dernière requête.
401	Unauthorized	Identification nécessaire pour accéder à la ressource.
403	Forbidden	Identification refusée.
404	Not found	Ressource non trouvée.
410	Gone	Ressource absente, sans adresse de redirection connue.
500	Internal Server Error	Erreur interne au serveur.
503	Service Unavailable	Serveur temporairement indisponible.

7. Mise en cache et requêtes conditionnelles

- La mise en cache est un élément crucial pour la plupart des systèmes distribués. Elle a pour but d'améliorer les performances en évitant les requêtes inutiles et en réduisant le volume des données des réponses. HTTP dispose de mécanismes permettant la mise en cache et la vérification de l'exactitude des données du cache. Si le client décide de ne pas utiliser ce cache, il devra toujours demander les données, même si elles n'ont pas été modifiées depuis la dernière requête.
- La réponse à une requête de type GET peut contenir un en-tête **Last-Modified** indiquant la date de dernière modification de la ressource. La prochaine fois que l'agent utilisateur demandera cette ressource, il passera cette date dans l'en-tête **If-Modified-Since** : le serveur web (ou le proxy) la comparera alors à la date de dernière modification. Si celle envoyée par l'agent utilisateur est égale ou plus récente, le serveur renverra une réponse sans corps, avec un code d'état **304 Not Modified**. Sinon l'opération demandée sera réalisée ou transférée.
- Les dates peuvent être difficiles à manipuler et impliquent que les agents concernés soient, et restent, synchronisés : c'est le but de l'en-tête de réponse **ETag**, qui peut être considéré comme un **hachage MD5** ou **SHA1** de tous les octets d'une représentation – **si un seul octet est modifié, la valeur d'ETag sera différente**. La valeur Etag reçue dans une réponse à une requête GET peut, ensuite, être affectée à un en-tête **If-Match** d'une requête.

7. Mise en cache et requêtes conditionnelles

- **Exemple:**

La figure ci-dessous montre un exemple d'utilisation d'ETag. Pour obtenir une ressource livre, on utilise une action GET en lui indiquant l'URI de la ressource (GET/book/12345). Le serveur répondra par une représentation XML du livre, un code 200 OK et un ETag. La seconde fois que l'on demandera la même ressource et que l'on passera la valeur de cet ETag dans la requête, le serveur ne renverra pas la représentation de la ressource, mais uniquement une réponse avec un code **304 Not Modified** afin d'informer le client que cette ressource n'a pas été modifiée depuis son dernier accès.

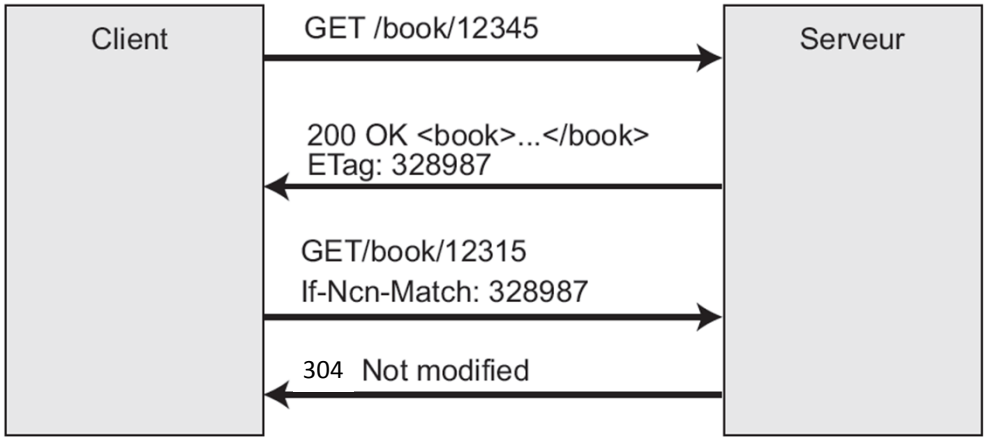


Figure 2. Utilisation du cache et du code 304.

7. Mise en cache et requêtes conditionnelles

- Les requêtes qui utilisent les en-têtes HTTP **If-Modified-Since**, **If-Unmodified-Since**, **If-Match**, **If-None-Match** et **If-Range** sont appelées **requêtes conditionnelles**. Elles permettent d'économiser la bande passante et le temps de calcul (à la fois sur le serveur et sur le client) en évitant des allers et retours ou des transferts de données inutiles. Généralement, ces en-têtes **If-*** sont surtout utilisés avec les requêtes GET et PUT.

Bibliographie

- Architectures réparties en Java: RMI, CORBA, JMS, sockets, SOAP, services web. *Annick Fron*. Dunod, 2007. ISBN: 2100511416, 9782100511419
- JAX-RS: Java™ API for RESTful Web Services. *Pavel Bucek and Santiago Pericas-Geertsen*. Version 2.1 Public Review, April 18, 2017. Oracle Corporation.
- Java EE6 et GlassFish 3. *Antonio Goncalves.* Editeur: Pearson, 2010. ISBN: 978-2-7440-4157-0
- SOA , microservices et API management : Le guide de l'architecte des SI agiles. *Fournier-Morel Xavier, Grojean Pascal, Plouin Guillaume*. Editeur: Dunod. Publication: 2017. ISBN: 978-2-10-076730-4.
- HTML 5 - Une référence pour le développeur web - . *Rodolphe Rimelé*. Edition: Eyrolles, 2013. ISBN: 9782212136388
- XML Cours et exercices. *Alexandre Brillant.* Edition: Eyrolles, 2007. ISBN : 978-2-212-12151-3