Architecture SOA et Web Services

Chapitre 2
RESTful web service

Raoudha Souabni
Fahima Ben Guirat

Contact:
souabni_raoudha@yahoo.fr
fahima.benguirat@gmail.com

ISTIC

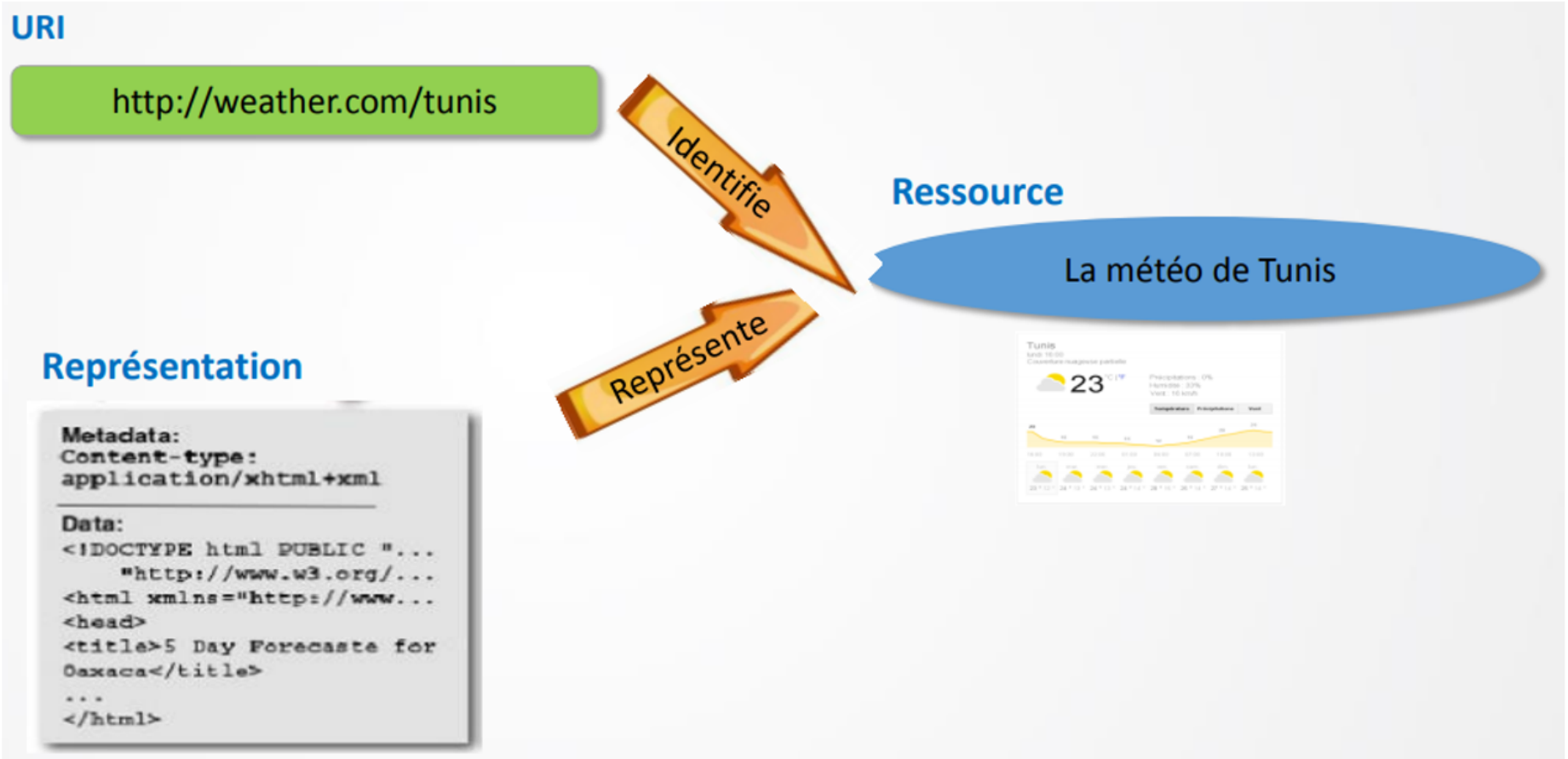
Plan

Plan

Rappel

- **REST = REpresentational State Transfer**
- REST est un style d'architecture permettant de construire des applications (Web, Web Service, ...).
- Il s'agit d'un ensemble de conventions et de bonnes pratiques à respecter et non d'une technologie à part entière.
- L'architecture REST utilise les spécifications originelles du protocole HTTP, plutôt que de réinventer une surcouche (comme le fait SOAP par exemple).

Principe


URI
http://weather.com/tunis
Identifie
Ressource
La météo de Tunis
Représente
Représentation
Metadata:
Content-type:
application/xhtml+xml
Data:
<!DOCTYPE html PUBLIC "...
"http://www.w3.org/...
<html xmlns="http://www...
<head>
<title>5 Day Forecaste for
Oaxaca</title>
...
</html>

Principe

Principes des Restful Web Service :

- Une ressource
- Un identifiant de ressource
- Une représentation de la ressource :
  - donne une vue sur l'état de la ressource
  - informations transférées entre le client et le serveur Exemples : XML, Text, JSON, ...
- Interagir avec les ressources en utilisant les requêtes HTTP

Principe

Principes des Restful Web Service :

- Chaque **'méthode'** ou **'service'** est considéré(e) comme une **ressource**
- Chaque ressource est identifiée par une **URI** (Uniform Resource Identifier)
- Utiliser les **verbes HTTP** existants (**GET**, **POST**, ...) plutôt que d'invoquer directement la méthode du web service
- La représentation de la ressource est échangée entre le client et le serveur en fonction de la méthode HTTP utilisée

➡ Consommer un WebService REST revient à appeler une simple URI en utilisant une méthode HTTP (Post ou Get ...)

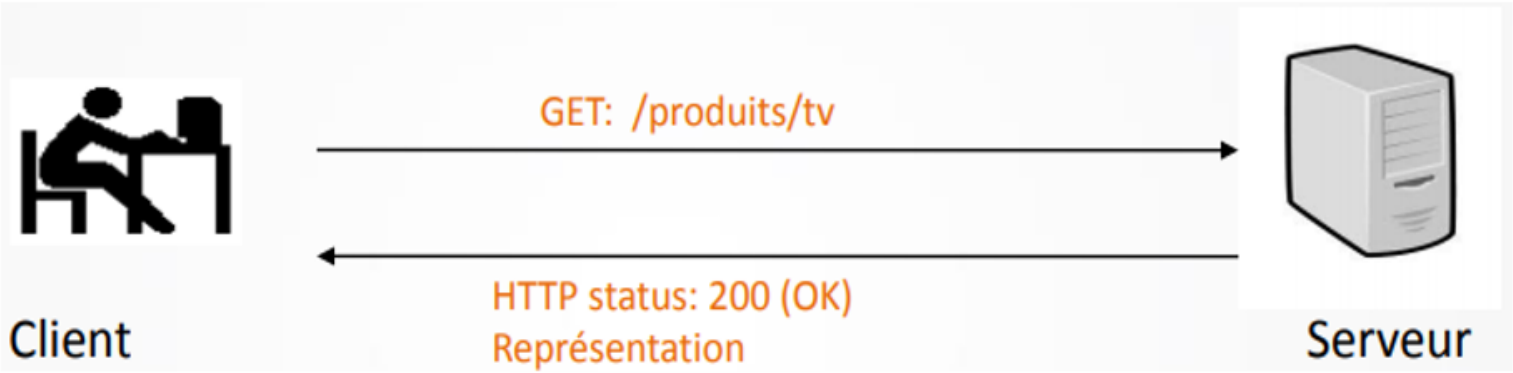
Verbes HTTP

- Ils servent à manipuler une ressource.
- HTTP définit, entre autres, quatre principaux verbes pour la manipulation des ressources :

Verbes	Rôle
GET	obtenir une représentation de la ressource
POST	créer une ressource
PUT	modifier une ressource en fournissant sa nouvelle représentation
DELETE	supprimer une ressource

Verbes HTTP et Représentation

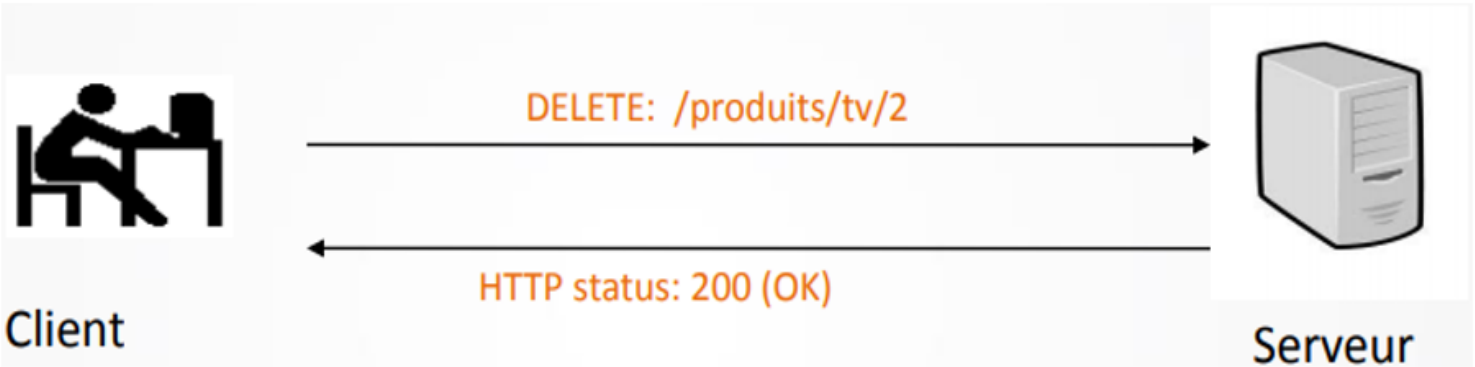
- Méthode GET fournit la représentation de la ressource



- Méthode POST crée une ressource

Verbes HTTP et Représentation

- Méthode DELETE supprime une ressource



- Méthode PUT met à jour une ressource

REST WS vs SOAP WS

	Services Web étendus SOAP	REST
	Exposition des opérations	Exposition des ressources
Protocole de communication	SOAP	HTTP
Protocole de transport	HTTP , autres	HTTP
Description des interfaces	WSDL	WADL
Format des données	XML	XML, Text, JSON

Description de services Web REST

- WADL (Web Application Description Language)
- Un langage de description XML de services de type REST
- Plus simple que WSDL (pas de séparation entre interface et binding)
- Permet une description de services par éléments de type : ressource, méthode, paramètre, requête, réponse.

Plan

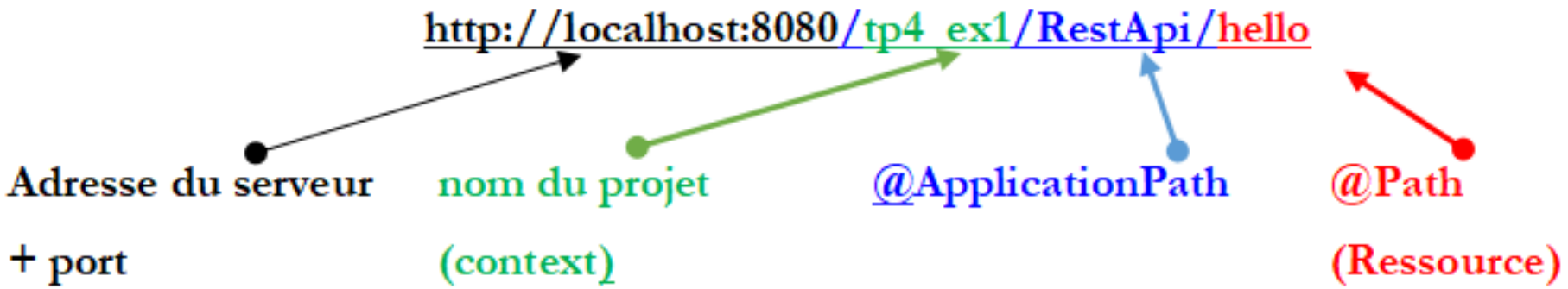
- Contrairement aux Services Web SOAP, il n'y a pas de possibilité de développer un service REST a partir du fichier de description WADL
- Seule l'approche **ascendante** **Botton - Up** est disponible :
  - Créer et annoter une classe java
  - Compiler et déployer
  - Tester
- Possibilité d'accéder au fichier de description WADL qui est généré automatiquement par JAX-RS

- L'implémentation des Services Web REST repose sur l'utilisation de classes Java et d'annotations
- JAX-RS est l'API conçue pour implémenter des Web Services RESTful
- Le package jakarta.ws.rs contient les annotations JAX-RS.
- Les annotations JAX-RS permettent de déclarer des classes de ressources et les types de donnée qu'elles prennent en charge.

Les annotations

@ApplicationPath

- Identifie le chemin de l'application qui sert d'URI de base pour toutes les ressources qui implémentes les services web.
- Elle doit être utilisée au niveau de la classe de configuration
- La classe de configuration est une classe qui étend la classe **jakarta.ws.rs.core.Application**.



Les annotations

@ApplicationPath

```
package tn.istic.rest;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("rest")
public class RestActivator extends Application{

}
```

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/hello")
public class HelloRessource {
    @GET
    @Produces("text/plain")
    public String sayHello(){
        return("Hello from JAX-RS");
    }
```

Les annotations

@Path

- Indique le chemin URI de la ressource
- Utilisée au niveau de la **classe** ainsi que ses **méthodes**
- Il est **obligatoire** d'annoter une classe de ressource avec l'annotation **@Path**
- L'annotation **@Path** sur une classe définit des ressources appelées **racines** (Root Resource Class)
- L'URI résultante est la concaténation de l'expression du *@Path de la classe* avec l'expression du *@Path de la méthode*

Les annotations

@Path

- La valeur définie dans @Path ne se limite pas seulement aux expressions constantes
- Possibilité de définir des expressions plus complexes appelées **Template Paramètres**
- Pour distinguer une expression complexe dans la valeur du **@Path**, son contenu est délimité par { ... }

```
@Path("{login}")
```

- Possibilité également de mixer dans la valeur de **@Path** des expressions constantes et des expressions complexes

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

Les annotations

Méthodes HTTP : @GET, @POST, @PUT, @DELETE

- L'annotation des méthodes Java permet de traiter les requêtes HTTP suivant le type de méthode (GET, POST, ...)
- Les principales annotations qu'on va étudier sont les suivantes **@GET**, **@POST**, **@PUT**, **@DELETE**
- Ces annotations ne sont utilisables que sur des méthodes Java
- Le nom des méthodes Java n'a pas d'importance puisque c'est l'annotation employée qui précise où se fera le traitement
- Des opérations CRUD sur des ressources sont réalisées au travers des méthodes HTTP

Les annotations

Paramètres de requête

- JAX-RS fournit des annotations pour extraire **les paramètres** d'une requête
- Elles sont utilisées sur **les paramètres des méthodes des ressources** pour réaliser l'injection du contenu
- Exemples d'annotations disponibles :
  - **@PathParam** : Permet d'extraire les valeurs des *Template Parameters*
  - **@QueryParam** : Permet d'extraire les valeurs des paramètres de requête
  - **@FormParam** : Permet d'extraire les valeurs des paramètres d'un formulaire

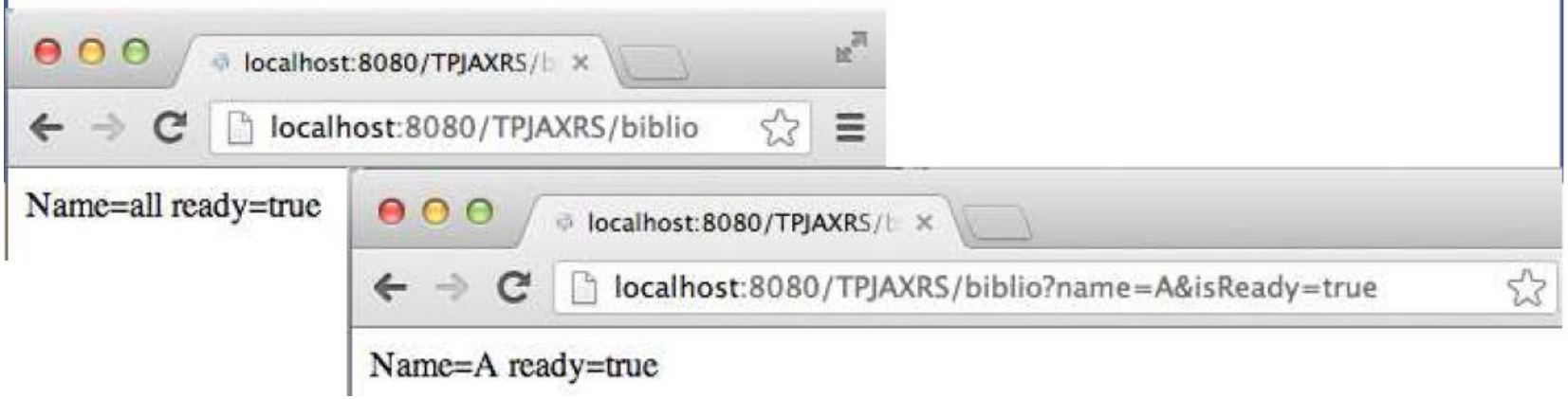
Les annotations

L'annotation **@PathParam** est utilisée pour extraire les valeurs des paramètres contenues dans les 'Template Parameters'

Les annotations

L'annotation **@QueryParam** est utilisée pour extraire les valeurs des paramètres contenues dans une requête

```
@GET
public String getQueryParameterBook(
        @DefaultValue("all")@QueryParam("name")String name,
        @DefaultValue("true")@QueryParam("isReady")boolean isReady ){
         return "Name="+name+" ready="+isReady;
        }
```



Les annotations

Les données transmises via un formulaire HTML peuvent être récupérées comme arguments des méthodes de la ressource JAX-RS grâce à l'annotation **@FormParam**.

```
<FORM action="http://example.com/customers" method="post">
    <P>
    First name: <INPUT type="text" name="firstname"><BR>
    Last name: <INPUT type="text" name="lastname"><BR>
    <INPUT type="submit" value="Send">
    </P>
 </FORM>
```

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

   @POST
   public void createCustomer(@FormParam("firstname") String first,
                              @FormParam("lastname") String last) {
     ...
   }
}
```

Les annotations

Representations : **@Consumes**, **@Produces**

- L'annotation **@Consumes** est utilisée pour spécifier le ou les types qu'une méthode d'une ressource peut accepter
  `@Consumes(value={MediaType.TEXT_HTML})`
- L'annotation **@Produces** est utilisée pour spécifier le ou les types qu'une méthode d'une ressource peut produire
  `@Produces(value={MediaType.APPLICATION_JSON,`
  `MediaType.APPLICATION_XML})`
- Possibilité de définir un ou plusieurs types
- Ces annotations peuvent être portées sur une classe ou sur une méthode

Plan

Manipuler l'objet Response

Supposons que nous allons implémenter un service qui :

- retourne la liste de tous les livres d'une bibliothèque en format json
- retourne un message indiquant que la bibliothèque est vide si aucun livre n'existe dans la bibiliothèque

$\longrightarrow$ Le même service retourne des réponses de types différents

$\longrightarrow$ Chaque réponse nécessite un traitement différent côte client

Besoin : Personnaliser la réponse

Techniquement : Utiliser l'objet Response

Manipuler l'objet Response

1- Paraméter la réponse avec :

- Un code de statut
  - Exemples : code 200, 404, 500
  - Techniquement : utiliser la méthode **status(int status)**
    $\longrightarrow$ Permet de céer un constructeur de réponse **ResponseBuilder** ayant le statut fourni en paramètre
    Exemple code 200 : Response.status(Status.OK)
    Exemple code 404 : Response.status(Status.NOT_FOUND)
    Exemple code personnalisé : Response.status(222)
- Une entité à retourner :
  - Techniquement : utiliser la méthode **entity(Object entity)** de la classe **ResponseBuilder**
  - Exemple : ResponseBuilder.entity("Erreur de connexion")

Manipuler l'objet Response

Classes des codes de statut :

- Les réponses informatives (100 - 199),
- Les réponses de succès (200 - 299),
- Les messages de redirection (300 - 399),
- Les erreurs du client (400 - 499),
- Les erreurs du serveur (500 - 599).

Manipuler l'objet Response

2- Construire la réponse :

- Techniquement : **ResponseBuilder.build()**
- Permet de créer un objet **Response** à partir de l'objet **ResponseBuilder**

Exemple :

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response getAll()
{
    if(liste.size()==0)
        return Response.status(Status.NO_CONTENT).entity(liste).build();
    else
       return Response.ok().entity(liste).build();
}
```

Plan

- JAX-RS Client API permet d'implémenter un client HTTP d'un service web Rest.
- Pour tester un service web Rest, il faut :
  1. Initialiser l'objet Client HTTP
  2. Identifier la ressource distante web service via son URI
  3. Préparer la requête avec la possibilité d'envoyer des données vers le serveur
  4. Invoquer la méthode HTTP appropriée
  5. Traiter la réponse retournée par le serveur

Initialisation de l'objet Client HTTP

- L'interface **Client** est l'élément d'entrée principal de l'API client JAX-RS. Elle permet de gérer les connexions avec le serveur.
- La classe **ClientBuilder** est utilisé pour créer des objets de type Client.
- **Techniquement** :

  - *Client client = ClientBuilder.newClient();*

  ou bien

  - *Client client = ClientBuilder.newBuilder().build();*

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;

public class ExempleClient {

  public static void main(String[] args) {
    Client client = ClientBuilder.newClient();
    // ...
  }
}
```

Identification de la ressource distante

- Le web service est une ressource distante identifiée par son URI
- L'interface **WebTarget** est utilisée pour configurer la ressource distante cible à invoquer identifiée par son URI.
- La méthode **target()** de l'interface **Client** permet de créer l'objet **WebTarget**
- **Techniquement** :

  WebTarget target = client.target("URI")

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;

public class ExempleClient {

  public static void main(String[] args) {
    Client client = ClientBuilder.newClient();
    WebTarget target = client.target("http://www.server.net/person");
    // ...
  }
}
```

Préparation des requêtes

Une fois l'objet WebTarget est créé, il est possible de l'utiliser pour préparer les requêtes HTTP via l'une des méthodes request() suivantes :

- **request()**
  Commence à créer une requête vers le service web ciblé.
- **request(String... acceptedResponseTypes)**
  Commence à créer une requête vers le service web ciblé en indiquant les types de médias de réponse acceptés.
- **request(MediaType... acceptedResponseTypes)**
  Commence à créer une requête vers le service web ciblé en indiquant les types de médias de réponse acceptés.

Invoquer une méthode HTTP - GET

- Après avoir créé la requête, on peut invoquer une méthode HTTP (GET, POST, ...) spécifique pour récupérer une réponse du serveur.
- Les requêtes **GET** ont deux types :
  - **<T> T get(Class<T> responseType)**
    ⟶ Permet de convertir les données d'une réponse HTTP réussie en un type Java spécifique

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;

public class ExempleClient {

  public static void main(String[] args) {
    Client client = ClientBuilder.newClient();
    WebTarget target = client.target("http://www.server.net/person");
    Person person = target.request().get(Person.class);
    // ...
  }
}
```

Invoquer une méthode HTTP - GET

- **Response get()**
  ⟶ Retourne un objet **Response**. Il s'agit de la même classe **Response** que celle utilisée côté serveur.
  ⟶ Permet un contrôle plus précis de la réponse HTTP côté client (gestion des cas de succès et des cas d'échec)

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://www.server.net/person");
Response resp = target.request().get();
if (resp.getStatus() == 200) {
    // Gestion du cas de succès 200
}
else
{
    //Gestion des autres cas
}
```

Invoquer une méthode HTTP - POST/PUT

- Les requêtes de type **PUT** et **POST** sont basées sur la soumission d'une représentation au serveur.
- Ces requêtes ont des types de méthode similaires à **GET** mais spécifient un paramètre **Entity** à envoyer vers le serveur :

- Les méthodes qui convertissent les données d'une réponse HTTP réussie en un type Java spécifique
  <T> T post(Entity<?> entity, Class<T> responseType)
  <T> T put(Entity<?> entity, Class<T> responseType)
- Les méthodes qui retournent un objet de type **Response**
  Response post(Entity<?> entity)
  Response put(Entity<?> entity)

Envoyer des données vers le serveur - Données de chemin

Pour envoyer des données de type **PathParam**, il est possible de :

- Indiquer directement les paramètres dynamiques au niveau de l'URI :

```
WebTarget target =
    client.target("http://server.tn/RestAPI/voyage/Tunisie/France");
```

- Etendre l'URI de base avec les paramètres dynamiques :

```
WebTarget base = client.target("http://server.tn/RestAPI/voyage");
WebTarget target = base.path("/Tunisie/France");
```

- Utiliser un URI template :

```
WebTarget target =
    client.target("http://server.tn/RestAPI/voyage/{VilleDep}/{VilleDest}")
          .resolveTemplate("VilleDep", "Tunisie")
          .resolveTemplate("VilleDest", "France");
```

Envoyer des données vers le serveur - Données de requête

Pour envoyer des données de type **QueryParam**, il faut utiliser la méthode **queryParam()**

```
WebTarget target = client.target("http://server.tn/RestAPI/voyage");
target = target.queryParam("id", 3);

//URI résultante : http://server.tn/RestAPI/voyage?id=3
```

Envoyer des données vers le serveur - Formulaire POST

Pour envoyer des données de type **FormParam**, il faut :

- Utiliser la classe **Form** pour créer le formulaire à envoyer
- Utiliser la classe **Entity** et la méthode **Entity.form()** pour créer une entité de type formulaire "**application/x-www-form-urlencoded**".

```
WebTarget target =  client.target("http://server.tn/RestAPI/voyage");
Form formulaire = new Form();
formulaire.param("identifier", "2");
formulaire.param("villeDep", "Egypte");
formulaire.param("villeDest", "Tunisie");
formulaire.param("dateVoyage", "11/11/2022");
Response resp = target.request().post(Entity.form(formulaire));
```

Envoyer des données vers le serveur - JSON

Pour envoyer des données de type **JSON**, il faut :

- Utiliser la classe Java spécifique pour créer l'objet Java à envoyer
- Utiliser la classe **Entity** et la méthode **Entity.json()** pour convertir l'objet Java créé en JSON et créer une entité de type "**application/json**".

```
Voyage voyage = new Voyage(2,"Egypte", "Tunisie","15/11/2022");
Response response = client.target("http://server.tn/RestAPI/voyage")
                    .request()
                    .put(Entity.json(voyage));
```

Traitement de la réponse Response

Principales méthodes de la classe **Response** :

- int getStatus()
  $\longrightarrow$ Permet de récupérer le code HTTP de statut
- boolean hasEntity()
  $\longrightarrow$ Permet de vérifier si une entité est disponible dans la réponse.
- boolean bufferEntity()
  $\longrightarrow$ Permet de mettre en mémoire tampon les données de l'entité du message.
- $< T >$ T readEntity(Class$< T >$ entityType)
  $\longrightarrow$ Permet de lire le flux d'entrée de l'entité de message en tant qu'instance du type Java spécifié

Traitement de la réponse Response

Exemple :

```
Response response = client.target("http://commerce.com/customers/123")
                         .request()
                         .get();
try {
   if (response.getStatus() == 200) {
       if (response.hasEntity()) {
              response.bufferEntity();
              Customer customer = response.readEntity(Customer.class);
       }
   }
} finally {
  response.close();
}
```