FORMATION

Développer des webservices en Java

21/ 07/ 2023

WebServices

Communications entre systèmes informatiques

- L’informatique connecte dès ses débuts ses machines entre elles pour paralléliser ou séquencer les informations, puis pour partager des informations entre services ou entreprises.
- Si des machines s’échangent des informations en pouvant les comprendre lors de l’émission et lors de la réception, l’échange d’informations devient un **protocole**.
- De nombreux protocoles sont utilisés pour faire communiquer des machines entre elles, depuis au moins les années 1960. ARPANET (ancêtre d’Internet) démarre en 1969.
- Une partie des standards sont disponibles ici : https://fr.wikipedia.org/wiki/Mod%C3%A8le_OSI#Quelques_protocoles , mais parfois les entreprises développent des protocoles propriétaires.

WebServices

Le Web en quelques mots

Le Web définit les objets suivants avec des spécifications (RFC) :

- Un hôte, ordinateur servant des ressources.
- Une ressource : une entité informatique (texte, image, vidéo…)
- HTTP : HyperText Transfer Protocol un protocole de communication utilisé pour transférer des ressources d’un hôte vers un client.
- URL : (Uniform Resource Locator) une chaîne de caractères définissant l’emplacement d’une ressource.
- HTML : (HyperText Markup Language) : décrit un document qui contient des hyperliens (entre autres).
- Hyperlien : un élément dans une ressource associée à une URL. La ressource derrière l’URL n’a pas connaissance des hyperliens qui lui sont rattachés.

Historique des WebServices (le Web)

- Le World Wide Web est inventé en 1989 (Tim Berners Lee au CERN).
- Le premier navigateur graphique ‘grand public’ arrive en 1993 (Mosaic).
- Le W3C est créé dans la foulée pour normaliser les RFC, et donc les protocoles liés à Internet.
- Il y a en 1993 500 serveurs Web grand public.
- Fin 1994, ils sont 10 000 ...
- ...232 millions en 2010.

WebServices

Historique des WebServices

- Dès les années 2000, il est clair qu’un protocole de communication est disponible sur de nombreux sytèmes et que la tendance ne va pas s’inverser.
- Les infrastructures réseaux sont aussi disponibles, que ce soit au niveau local à une entreprise, ou au niveau international.
- Les normes sont stables et le W3C fait évoluer ou stabilise ces normes indépendamment d’une entreprise ou d’un état.
- Il est dès lors extrêmement tentant d’utiliser ce protocole pour faire communiquer non pas des humains avec des machines, comme avec un navigateur, mais pour faire communiquer des machines entre elles.

WebServices

Définition d’un WebService

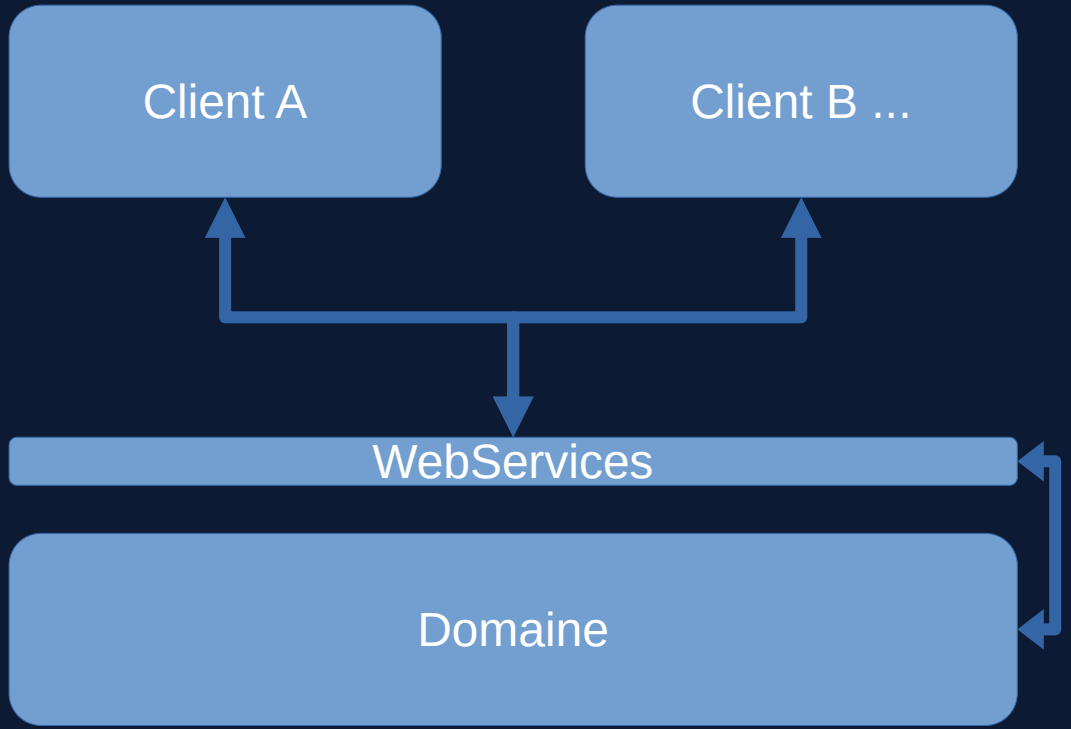
- Un WebService est un service qu’offre un serveur, disponible sur le Web.
- Il se base donc sur les normes du W3C, dont HTTP.
- Il peut obéir à un des standards, type :
  - SOAP (complet mais complexe, et de moins en moins à la mode).
  - REST (simple, de plus en plus utilisé).

https://fr.wikipedia.org/wiki/Service_web#

Réutilisation et granularité

- Les WebServices étant pensés comme des services liés à des ressources du Web, il faut découper le modèle en ‘ressources’.
- Ce découpage fonctionnel dépend du domaine à découper, ainsi que de l’interface que l’on veut présenter.
- Il peut être vu comme une surcouche du domaine.
- Les ressources doivent être le plus souvent indépendantes du client qui en aura besoin. Ceci rend la couche de WebServices réutilisable.

Réutilisation et granularité



Cycle de vie d’un Web Service

- Le WebService doit être pensé comme un “contrat” auquel les clients vont adhérer pour communiquer avec vos systèmes. Il vaut donc mieux le :
  - Documenter (Texte, Word, mais il vaut mieux utiliser des spécifications comme Open API).
  - Versionner. Toute évolution non rétrocompatible du contrat devra figurer dans une nouvelle version : les WebServices devront être relivrés.
- Le WebService se base sur les spécifications du W3C. Il vaut mieux obéir à ces spécifications au maximum pour que les clients aient le moins de surprise possible lors de leur connexion aux WebServices.
- Une fois le couplage fait avec un ou plusieurs clients, il faut maintenir ces WebServices, s’assurer de la qualité du service, et faire évoluer les contrats et/ou les WebServices.
- Les services sont accessibles à n’importe qui : il faut donc prévoir de quoi communiquer lorsque ces services sont débranchés (inscription des utilisateurs à un medium d’information, création d’un service d’information sur les WebServices, …).

https://www.openapis.org/

WebServices

Resource Oriented Architecture (ROA)

- ROA définit toute architecture basée sur REST.

https://en.wikipedia.org/wiki/Resource-oriented_architecture

Structures de données échangées

- Les normes du W3C permettent d’échanger tout type de données grâce à des types MIME (Multipurpose Internet Mail Extensions), définis dans des RFC (6838 et 4289).
- Elles permettent aussi de spécifier un encodage au besoin (type UTF-8 pour du texte).
- Il est donc possible d’échanger du texte, des musiques, des vidéos, des documents au format propriétaire (type MicrosoftOffice) par WebServices.
- Par contre, dès qu’un document contient des données formattées avec un ‘langage’ propre au WebService, il faut documenter ce langage. Cela peut être via un XSD, ou tout autre document.

https://fr.wikipedia.org/wiki/Multipurpose_Internet_Mail_Extensions

JSON

JSON

Historique

- Langage équivalent à XML créé entre 2002 et 2005.
- Normé depuis 2004. Une RFC le spécifie (la 8259).
- Utilisé pour Javascript, puis pour des fichiers de configuration ou d’échange (remplace le XML, car plus simple).

https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON

JSON

JSON

- JSON est un format de données semblable à la syntaxe des objets JavaScript, mais peut être utilisé indépendamment de ce langage. De nombreux autres langages de programmation disposent de fonctionnalités permettant d'analyser la syntaxe du JSON et d'en générer.
- Le JSON se présente sous la forme d'une chaîne de caractères (utile pour transmettre des messages via un protocole texte comme HTTP). Le JavaScript fournit un objet global JSON disposant des méthodes pour assurer la conversion entre les deux.

https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON

JSON

Types en JSON

Les types de base du format JSON sont les suivants :

- Chaîne de caractères : une séquence de 0 ou plus caractères Unicode. À l'instar des clés, elles sont obligatoirement entourées de guillemets (non interchangeables avec des apostrophes).
- Nombre : un nombre décimal signé qui peut contenir une part fractionnable ou élevée à la puissance (notation E). Le Json n'admet pas les nombres inexistants (NaN), et ne fait aucune distinction entre un entier et un flottant.
- Booléen : true ou false sont utilisés pour définir l'état du booléen ;
- Type null : une valeur vide, utilisant le mot clé null.
- Tableau: entouré par des crochets. Contient un ensemble d’objets du même type.
- Objet : entouré par des accolades. Contient un ensemble de types.

JSON

N’existe pas en JSON

- Les types Date ou Time.
- Les commentaires.

JSON

Exemple de JSON

```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ]
    },
    {
      "name": "Madame Uppercut",
      "age": 39,
      "secretIdentity": "Jane Wilson",
      "powers": [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ]
    }
  ]
}
```

JSON

Utilisation de JSON en JavaScript

- Ouvrir la page src/web/json/json-base.html avec un navigateur et observer le résultat.
- Ouvrir le fichier avec un éditeur de texte : la fonction JavaScript populate() fait un appel HTTP à https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json et utilise JSON.parse() pour transformer la chaîne de caractères présente dans la réponse du serveur, en objet JSON.
- Observer la méthode populateHeroes pour voir comment JavaScript accède à différents attributs d'un objet JSON, et rajouter dans l'attribut textContent de myPara1 l'identité secrète du héros, en s'aidant de la définition du message JSON.
- Bonus : créer myParam4 et y afficher le premier pouvoir du super-héros.

JSON

Méta schéma JSON

- Contrairement à XML, de base, JSON ne propose pas de “méta-modèle”, un document spécifiant un modèle.
- Néanmoins, une initiative nommée JSON Schema existe, elle permet de définir un schéma JSON (ou méta modèle). Ce meta-modèle valide un document JSON.

https://json-schema.org/

JSON

Méta schéma JSON : spécification

```
{
    "$schema": "http://json-schema.org/draft-04/schema#", ← Ce schéma est écrit en utilisant la spécification
draftv4 de json schema
    "title": "Robot", ← Titre du schéma
    "description": "Description d'un robot", ← Description du schéma
    "type": "object", ← Le type du premier objet est obligatoirement object
    "properties": {
        "id": {
            "description": "L'identifiant unique",
            "type": "number", "minimum": 1
        }, ← On définit une première proprieté de l'objet  : id de type number
        "dateConstruction": {
            "type": "string", format="date"
        }, ← une autre de type String, formattée comme une date
        "software": {
            "description": "Equipement logiciel",
            "type": "object", ← Software est un objet, qui contient lui même d'autre propriétés...
            "properties": {
                    "name": {
                         "description": "Nom du logiciel",
                         "type": "string","maxLength": 20
                     }, ← … un nom et …
                "frameworks": {
                    "type": "array",
                    "items": {
                        "description": "Liste des frameworks",
                        "type": "string"
                    }
             }← …  un tableau de strings
            }
        }
    }
}
```

JSON

Méta schéma JSON : exemple

Le document ci-dessous correspond à la spécification à droite :

```
{
	"prenom":"Jean",
	"nom":"Dupont",
	"dateDeNaissance":"2002-12-12",
	"adresse":{
		"nomVoie":"4 RUE TABAGA",
		"codePostal":"12123",
		"ville":"VILLEFRANCHE",
		"pays":"FRANCE"
	}
}
```

```
{
  "type": "object",
  "properties": {
    "prenom": { "type": "string" },
    "nom": { "type": "string" },
    "dateDeNaissance": { "type": "string", "format":
"date" },
    "adresse": {
      "type": "object",
      "properties": {
        "nomVoie": { "type": "string" },
        "codePostal": { "type": "string" },
        "ville": { "type": "string" },
        "pays": { "type" : "string",
            "enum": ["FRANCE", "BELGIQUE", "SUISSE"] }
      }
    }
  },
  "required": ["nom","prenom"]
}
```

https://json-schema.org/

JSON

Méta schéma JSON : remarques

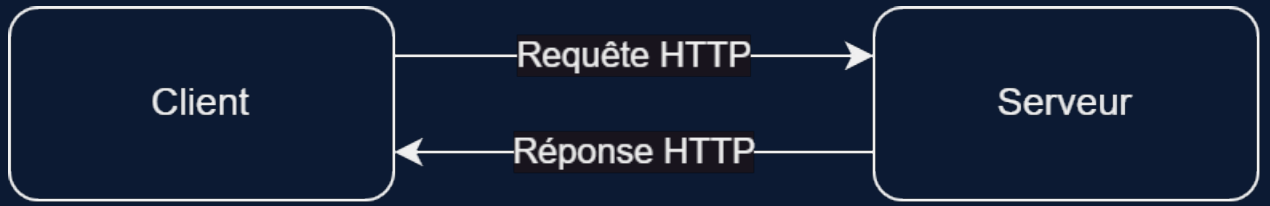
- Un méta schéma JSON permet de renforcer le typage d’un document JSON avec :
  - Des champs obligatoires
  - Des Types Date, Time, Date-Type (ou plutôt des String formattées comme tels)
  - Des enums
  - Des valeurs min et max pour les nombres
  - Etc …
- Attention, ce schéma est très utilisé, mais pas encore standardisé.

https://json-schema.org/draft/2020-12/json-schema-validation.html

WebServices et Java

Les webservices

Les webservices sont des fonctionnalités qu'un serveur Web offre à un ou plusieurs clients. Le serveur Web répond à des requêtes HTTP par des réponses HTTP. Un client et un serveur de WebServices communiquent donc par WebServices pour échanger des données.



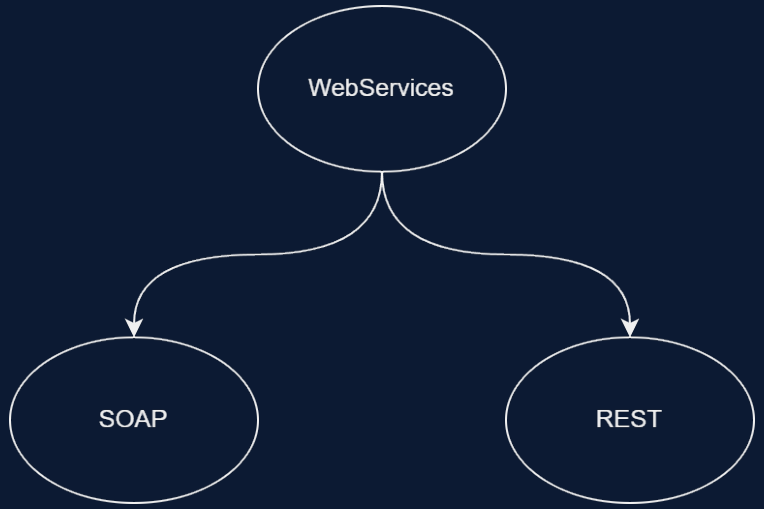
HTTP

- HyperText Transfer Protocol
- Codifie un échange entre deux applications :
- Via des requêtes et des réponses écrites en texte (lisible par un humain)
- Une requête contient :
- une ligne de commande (Commande, URL, version du protocole). Exemple GET /index.html HTTP/1.0
- Un ou plusieurs entêtes (Header) ex : Host: mysite.fr
- Un corps (Body). Ex : {‘id’ : 3, ‘name’:’MyCorp’}
- Une réponse contient un code réponse, un ou plusieurs en têtes, un corps (optionnel).
- Il est décrit par des RFC (Request For Comments) de l’IETF (Internet Engineering Task Force).

https://developer.mozilla.org/fr/docs/Web/HTTP

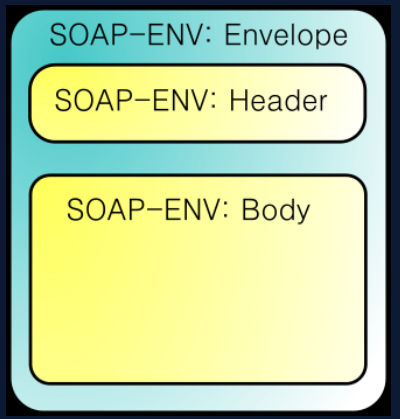
Types de webservices

HTTP est majoritairement utilisé en entreprise pour faire communiquer des humains avec des machines, mais aussi des machines entre elles, depuis le début des années 2000. Dès lors, une tentative a été faite de renforcer le protocole HTTP pour les communications entre machines : SOAP. SOAP étant (trop) complexe, le protocole REST lui a succédé dans de nombreux projets.



SOAP

- SOAP est un protocole bâti sur XML et généralement sur HTTP.
- Il définit la communication entre client et serveur en spécifiant la forme des messages. Un message est composé de :
  - une enveloppe qui définit la structure du message.
  - un entête qui contient des méta données
  - un corps qui contient les informations de requête et de réponse.



https://fr.wikipedia.org/wiki/SOAP

SOAP

- Ci-dessous un exemple de message SOAP pour requêter le prix de l'action AT&T (son identifiant est T) :

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:m="http://www.example.org">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice>
      <m:StockName>T</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

SOAP : critiques

En théorie, SOAP est complet. En pratique, il est complexe :

- SOAP est bien moins "simple" que son nom ne le suggère. Il est verbeux, et utilise le XML. De plus, il définit une surcouche de HTTP, mais oblige les utilisateurs à RE-définir un protocole au-dessus de SOAP pour créer leurs messages.
- SOAP utilise de facto HTTP, mais se veut indépendant du protocole. Toutes les possibilités de HTTP ne sont pas directement utilisables par SOAP.

REST : théorie

Representational State Transfer

Pour résumer :

- Une interface uniforme pour toutes les ressources (traduire par HTTP GET/POST/DELETE avec une URL bien formée, souvent en JSON)
- Client serveur (traduire par HTTP)
- Stateless (comme … HTTP)
- Cacheable (traduire par utiliser le paramètre Cache-Control de HTTP)
- Avec des serveurs interconnectés.
- Avec du code à la demande (optionnel, tellement que pas grand monde ne le fait).

Exemple de requête réponse REST

Ci-dessous une requête HTTP pour pour requêter le prix de l'action AT&T (son identifiant est T) :

```
GET /InStock/stock-prices/T HTTP/1.1
Host: www.example.org
Accept-Language: fr
```

- Et sa réponse transmise par le serveur :

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/json

{stock : ... (avec les 29769 octets de la réponse, comme promis !)
```

REST : pourquoi ça fonctionne ?

- Dans les années 2000, l'émergence des webservices a conduit les développeurs à tenter de formaliser ces interfaces. Ceci a conduit aux définitions de Webservices par WSDL. Le WSDL , bien défini, devait servir à générer des clients et des serveurs capables de se comprendre, par exemple en générant des classes de clients, serveurs et d'objets partagés. Ceci, plus la validation XSD à tous les étages, alourdissait le coût de développement des Webservices, au moment où les développeurs avaient besoin de les faire évoluer rapidement. REST, plus simple, plus proche du HTTP a eu la faveur des développeurs.
- Le document ci-dessous est un bon exemple de bonnes interfaces REST.

https://raw.githubusercontent.com/tfredrich/RestApiTutorial.com/master/media/RESTful%20Best%20Practices-v1_2.pdf

REST : URLs

Quelques URLs de requêtes REST ayant du sens :

- GET http://www.example.com/customers/12345 : récupère le client ayant l'identifiant 12345
- GET http://www.example.com/customers/12345/orders : récupère les commandes de l'utilisateur ayant l'identifiant 12345
- PUT http://www.example.com/customers/12345 : avec un body, met à jour le client 12345
- POST http://www.example.com/customers : avec un body, met à jour (ou crée) un client. L'identifiant est dans le body.
- POST http://www.example.com/customers/12345/orders : met à jour (ou crée) la commande du client 12345
- DELETE http://www.example.com/customers/12345 : supprime le client 12345

REST WebServices

HATEOAS

HATEOAS (Hypermedia as the Engine of Application State) est un lien hypertexte pour les WebServices

GET http://api.domain.com/management/departments/10


```
{
    "departmentId": 10,
    "departmentName": "Administration",
    "locationId": 1700,
    "managerId": null,
    "links": [
        {
            "href": "10/employees",
            "rel": "employees",
            "type" : "GET"
        }
    ]
}
```

REST WebServices

JSON

- JavaScript Object Notation
- Du XML en plus concis, sans XSD

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd
Street",
    "city": "New York",
    "state": null,
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
....
```

REST en JAVAEE

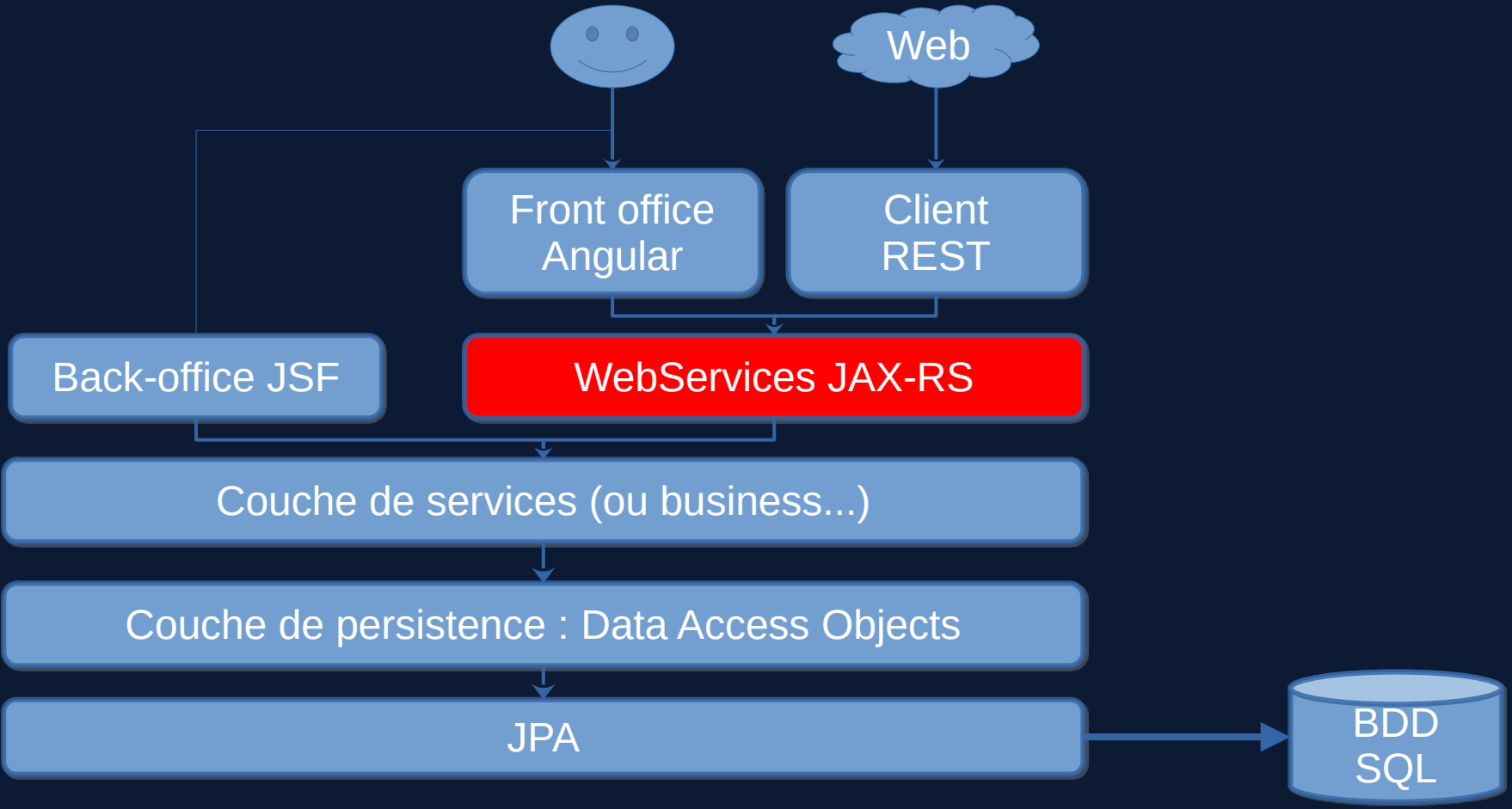
- JavaEE propose une API pour le REST : JAX-RS.
- L'implémentation officielle de JAX-RS est Jersey.
- Une bibliothèque de la fondation Apache implémente JAX-RS et l'étend même : CXF.

REST en Spring MVC

- Spring WebMVC permet de développer des Webservices REST avec Spring

REST WebServices

L’application “CRM”



JAX-RS : configuration

- La spécification est assez simple. Pour les cas classiques, un seul bean de configuration suffit pour JAX-RS :

```
@ApplicationPath("/rest/v1") ← Cette URL sera la base de toutes nos URLS (il faut toujors
versionner ses webservices)
public class ApplicationConfig extends Application {

}
```

JAX-RS : un premier service REST

- Créons un RestService pour notre Client
- Ajoutons la méthode GET pour récupérer la ressource selon son id

```java
@Stateless
@Path("/clients") ← Le nom de la ressource est très généralement au pluriel
@Produces("application/json") ← Format incontournable en REST
public class ClientRestService {

    @Inject
    ClientService clientService;


    @GET
    @Path("/{id}") ← L'URL finale sera : application+classe+méthode
    @Produces("application/json") ← On peut surcharger les comportements par annotations
    public ClientRestBean getclientById(@PathParam("id") String id) {← Correspond à {id}
        return toClientRestBean(this.clientService.findById(Integer.decode(id)));
    }
```

Outillons-nous pour tester

- Téléchargez un plugin pour votre navigateur (OpenRESTed pour Firefox), ou PostMan.
- Testez l'URL http://localhost:8080/crm/rest/v1/clients/9512 (spoiler, une belle 404 apparaît).

PostMan

- PostMan est à l’origine un client REST qui permet d’envoyer des requêtes HTTP.
- Il contient de nombreuses autres fonctionnalités : sauvegarde des requêtes, gestion des bibliothèques, des environnements, des paramètres ...
- Aujourd’hui, il propose de nombreux produits autour des “APIs”, terme commercial qui signifie : Webservices REST.

JAX-RS : la méthode POST

- Ajoutons la méthode POST.
- Celle-ci sauvegarde ou met à jour la ressource.
- Elle n’est pas idempotente (donc ne devrait pas être cachée). Ici deux appels à POST avec la même ressource sans id vont renvoyer deux ressources avec deux ids différents.

```
@POST
@Produces("application/json")
public ClientRestBean postClient(ClientRestBean clientRestBean) {
   if(clientRestBean == null) {
         return null;
   }
     return toClientRestBean(this.clientService.save(toclient(clientRestBean)));
}
```

Codes de retour

- Idéalement, un serveur HTTP retourne des réponses de type 2XX quand tout va bien,
- de type 4XX si le client a fait une erreur,
- de type 5XX si l’erreur est côté serveur.

  Il faut aussi idéalement que le corps du message contienne des informations intéressantes pour que le client puisse résoudre ses problèmes au plus vite.

https://www.rfc-editor.org/rfc/rfc7807

Codes de retour

- Il existe de nombreuses manières de gérer ces codes de retour.
- Celle vue en cours pour JAX-RS utilisera la classe Response, qui contient les éléments (Body, Headers ...) d'une réponse HTTP
- L'utilisation d'un Builder facilite la création des Responses.

```
@GET
@Path("/{id}")
public Response getclientById2(@PathParam("id") String id) {
     Client client = this.clientService.getById(Long.decode(id));
     if (client == null) {
          return Response.status(Response.Status.NOT_FOUND).
                    entity("Erreur").type(MediaType.TEXT_PLAIN).build();
     }
     return Response.ok(toRestDto(client), MediaType.APPLICATION_JSON).build();
}
```

Exercice sur la ressource client

- Créer le service ClientRestService
- Y ajouter les méthodes pour gérer les appels HTTP GET (par id), POST et DELETE (par id)

Validation des beans

Il est tout à fait possible de lancer la javax.validation dans JAX-RS :

```
@POST
@Produces("application/json")
public ClientRestBean postclient(@Valid ClientestBean clientRestBean) {
   if(clientRestBean == null) {
         return null;
   }
   return toClientRestBean(this.clientService.save(toClient(clientRestBean)));
}
```

Si clientRestBean contient des annotations de validation, des validations seront effectuées.

Validation des beans

Afin de transformer les exceptions en réponses HTTP, on peut utiliser un ExceptionMapper :

```
@Provider
public class ConstraintViolationExceptionMapper implements ExceptionMapper<ConstraintViolationException> {

@Override
public Response toResponse(final ConstraintViolationException exception) {
	return Response.status(Response.Status.BAD_REQUEST).entity(prepareMessage(exception)).type("text/
plain")
	.build();
}

private String prepareMessage(ConstraintViolationException exception) {
	StringBuilder msg = new StringBuilder();
	for (ConstraintViolation<?> cv : exception.getConstraintViolations()) {
		msg.append(cv.getPropertyPath() + " with value " + cv.getInvalidValue() + " " +
		cv.getMessage() +  "\n");
	}
	return msg.toString();
	}
}
```

Exercice sur la ressource clients

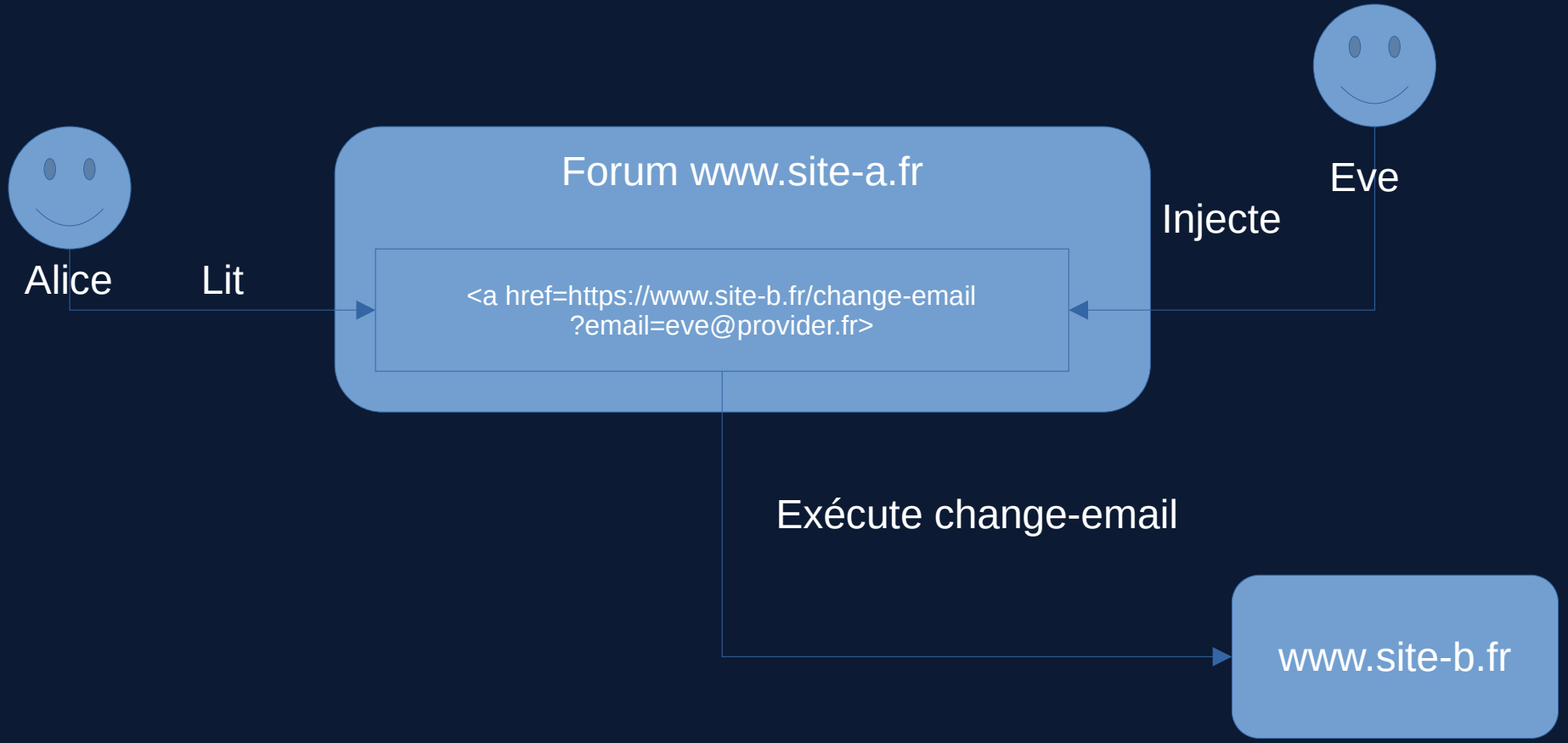
- Valider le fait que le nom soit non null et au minimum de 3 caractères.
- Renvoyer une erreur 400 dans le cas contraire

REST WebServices

TP Créer les ressources nécessaires au Front-office de Booking

- GET /clients/{id}
- POST /clients/{id}/tables
- GET /clients/{id}/tables
- GET /clients?name=…. (name peut être vide)
- GET /orders/{id}
- DELETE /orders/{id}
- POST /orders
- GET /orders (find all)

CORS : attaque CSRF



CORS : protection

- La protection Cross-origin Resource Sharing s’effectue au niveau du navigateur.
- Par défaut, celui-ci se base sur la politique : Same-origin policy : toute requête provenant de la page du site www.site-a.fr doit aller sur le même site.
- Le niveau de protection peut être réduit par les serveurs appelés, en fournissant des HEADERS sur les réponses HTTP.
- Le navigateur choisit d’envoyer la requête et de s’assurer que les headers de réponse sont bons (requête simple), ou envoie une requête OPTIONS pour récupérer les headers de réponse et voir s’ils acceptent le CORS.

CORS : configuration pour JAX-RS (pour la formation)

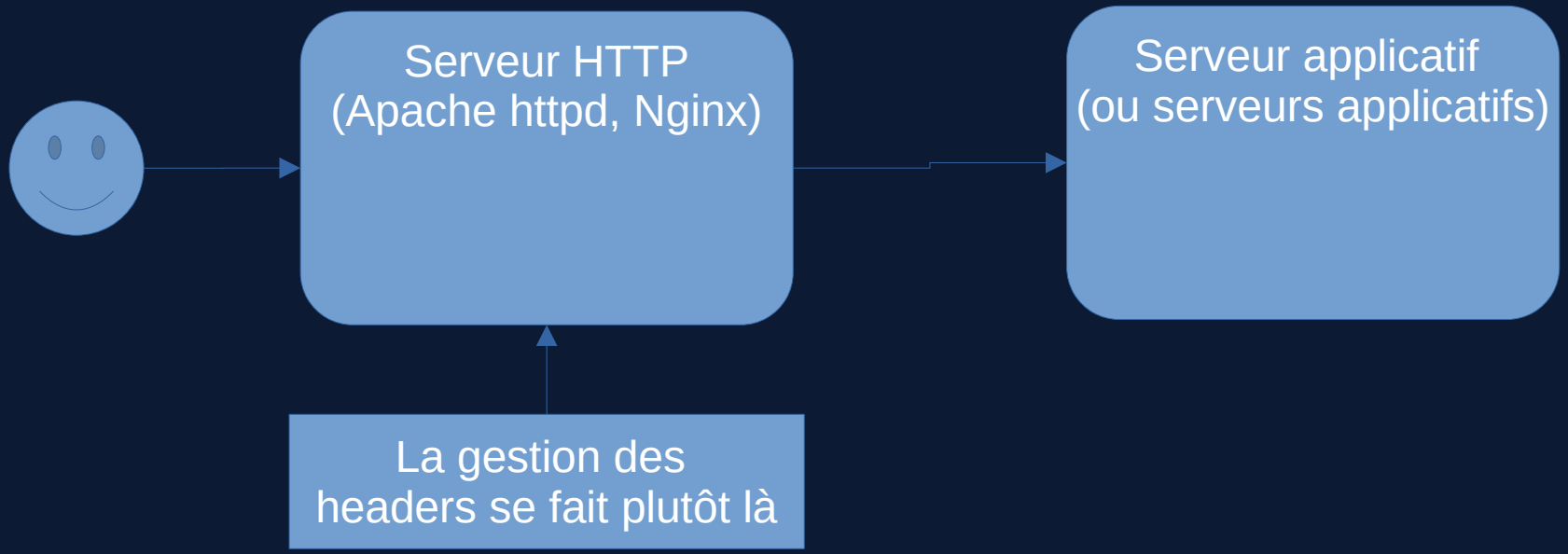
Créer une classe CorsFilter telle que suit. Celle-ci est bien trop permissive pour la production, mais suffira pour les tests.

```
@Provider
public class CorsFilter implements ContainerResponseFilter {

@Override
public void filter(ContainerRequestContext requestContext, ContainerResponseContext responseContext)
throws IOException {
   responseContext.getHeaders().add("Access-Control-Allow-Origin", "*");
   responseContext.getHeaders().add("Access-Control-Allow-Credentials", "true");
   responseContext.getHeaders().add("Access-Control-Allow-Headers", "origin, content-type, accept,
authorization");
   responseContext.getHeaders().add("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS, HEAD");
   }

}
```

CORS : configuration pour JAX-RS (en production)



Fin

Conclusion

Merci pour votre attention !