# Programmation d'Applications Web Avancées

## Cours 3
## Services Web REST
## JSON

kn@lri.fr

UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir

université PARIS-SACLAY

# Plan

# Service Web

Un service Web est un moyen pour une application d'exposer son API au moyen du protocole HTTP

Il existe plusieurs méthodologies :

- Web API : utilise différents standards tels que WSDL (fichier XML décrivant le service), SOAP (protocole avec des messages XML au dessus de HTTP), UDDI (annuaire de service), ...
- REST : Representational State Transfer. Approche plus simple basée sur des URL et les méthodes HTTP (GET, POST, PUT, DELETE)

On utilisera des services Web REST

# REST

Architecture issue de la thèse de Roy Thomas Fielding (2000, U. of Cal. Irvine)

Un service Web REST se caractérise de manière suivante :

- Une ou plusieurs URL servant de point d'entrée
- Un comportement implémenté pour les requêtes GET, POST, PUT, DELETE.

Les contraintes de méthode sont les suivantes

- GET : le traitement du serveur doit être sans effet (read-only)
- PUT/DELETE : le traîtement doit être idempotent (l'état exposé par le système ne doit pas varier après un appel à la méthode avec un paramètre donné)
- *stateless* chaque requête transporte l'état nécessaire, stocké côté client

Le contenu renvoyé par le service Web peut être quelconque, mais on privilégiera le JSON dans ce cours

# Classe utilitaire/Appel de service Web



La classe `HTTPUrlConnection` permet de créer des requêtes HTTP en Java (voir la Javadoc). Pour des requêtes simples (GET) on poura utiliser uniquement la classe `URL`.

# Exemples d'organisation URI/Méthodes HTTP (1)

Un site veut exporter une *collection* d'objets ainsi que des opérations sur ces derniers.
Url de collection : `http://site.com/api/collection/`

- GET : liste les objets, la taille de la collection, ...
- POST : ajoute un élément (des informations sur l'éléments à créer peuvent être données en paramètres de requête)
- PUT : remplace toute la collection par une autre
- DELETE : supprime la collection
- PATCH : inutilisé

Url d'un objet : `http://site.com/api/collection/object123`

- GET : récupère l'objet et ses méta-données
- POST : inutilisé
- PUT : remplace l'objet existant ou le crée
- DELETE : supprime l'objet
- PATCH : modifie l'objet

# Exemples d'organisation URI/Méthodes HTTP (2)

S'il y a une base de donnée sous-jacente alors :
Url de collection : `http://site.com/api/collection/`

- GET → `SELECT id FROM COLLECTION`
- POST → `INSERT INTO COLLECTION VALUES`
- PUT → `DROP TABLE; CREATE TABLE; INSERT`
- DELETE → `DELETE FROM COLLECTION`

Url d'un objet : `http://site.com/api/collection/object123`

- GET → `SELECT * FROM COLLECTION WHERE id=123`
- PUT → `DELETE/INSERT`
- DELETE → `DELETE FROM COLLECTION WHERE id=123`
- PATCH → `UPDATE … SET … WHERE id=123`

# *Stateless ?*



Dans une architecture REST, le serveur ne conserve *aucun état* spécifique à chaque client (i.e. pas de session, ni de cookies).

Chaque client est responsable de conserver son état localement.

Un mecanisme d'authentification/authorisation doit être mis en place

# Authentification

Il existe plusieurs méthodes pour s'authentifier auprès d'un service Web :

Basic Auth : Ajout du `user:password` dans un header HTTP `Authorization`

*Avantage* ⇒ simple

Inconvénient ⇒ peut sûr

API Key : Le client récupère auprès du serveur un secret qu'il ajoute à chaque requête (le secret peut être renouvelé fréquement). *Avantage* ⇒ simple

Inconvénient ⇒ peut sûr, non standard

OAuth : St *Avantage* ⇒ assez sûr, standardisé

Inconvénient ⇒ mise en place complexe

# Étude de cas

Application mobile de réservation de billets d'avion.

Une compagnie aérienne possède un serveur de réservation de billets d'avions, accessible par une interface Web standard ainsi que via un service Web

Un exemple de consommateur du service Web peut être une application mobile native (i.e. pas une qui affiche le site web mobile)

# Étude de cas : site Web



- Le client est associé à sa session HTTP
- L'état du client (page sur laquelle le client est, étape dans la procédure de réservation, ...) est stocké côté serveur
- Le client peut s'authentifier par login/mot de passe
- Le client peut modifier l'état du serveur (en créant une reservation par exemple)

# Étude de cas : Appli Mobile



- Le client possède une clé d'API
- L'état du client (page sur laquelle le client est, étape dans la procédure de réservation, ...) est stocké côté client uniquement
- Le client peut modifier l'état du serveur (en créant une reservation par exemple), il doit avoir une clé d'API valide

# Format de réponse



REST privilégie les réponses en mode texte. Le *type mime* de la réponse doit être spécifié dans les en-tête HTTP :

- text/xml (XML lisible par un humain)
- application/xml (XML complexe)
- application/json
- text/plain (fichier texte)
- text/csv (fichier CSV)
- ...

# Plan

# JSON (ou, le nouveau XML)



Il est souvent utile de pouvoir échanger de l'information « structurée » entre applications

- Chargement et production simplifiée (parseur/pretty-printer générique)
- Validation stricte possible (notion de bonne formation, schéma)

Solutions actuelles

- Format texte ad-hoc
- Format texte structuré (CSV)
- Format binaire ad-hoc
- XML
- *JSON* : JavaScript Object Notation (« Jay-zon »)

# JSON : syntaxe



Une valeur JSON est représenté par un sous-ensemble de la syntaxe Javascript pour les objets.

- *null* : la constante `null`
- Booléens : les constates `true` ou `false`
- Nombres : les nombres au format IEEE-759
- Les chaînes de caractères : délimitées par des `"` (obligatoirement), avec les séquences d'échappement usuelles (`\n`, ...)
- Les tableaux : suite de valeurs séparées par des virgules, entre `[ ]`
- Des objets : les nom propriétés sont des *chaines* de caractères (séparés par des virgules)

```
{ "nom" : "Nguyen",
  "prénom" : "Kim",
  "cours": [ "PAWA",  "LCD", "POS" ],
  "full time" : true,
  "age" : 3.6e1,
  "hobby" : null  }
```

# JSON : syntaxe (suite)



- Pas de syntaxe pour des commentaires
- Une propriété est n'importe quelle chaine syntaxiquement valide
- Les tableaux peuvent contenir des types différents
- Les blancs en dehors des chaînes ne sont pas significatifs
- Les chaînes ne peuvent pas être sur plusieurs lignes

# API Java pour JSON



Aucune API dans la bibliothèque standard Java (proposée initialement pour Java 9 puis abandonnée)

Une spécification « standard » fait partie de JEE 7 : *JSON-P*

Plusieurs implémentations de la spécification

(en pratique, ça se traduit par un `.jar` en plus dans le projet).

# `Factory` ***design pattern*** **(rappels)**



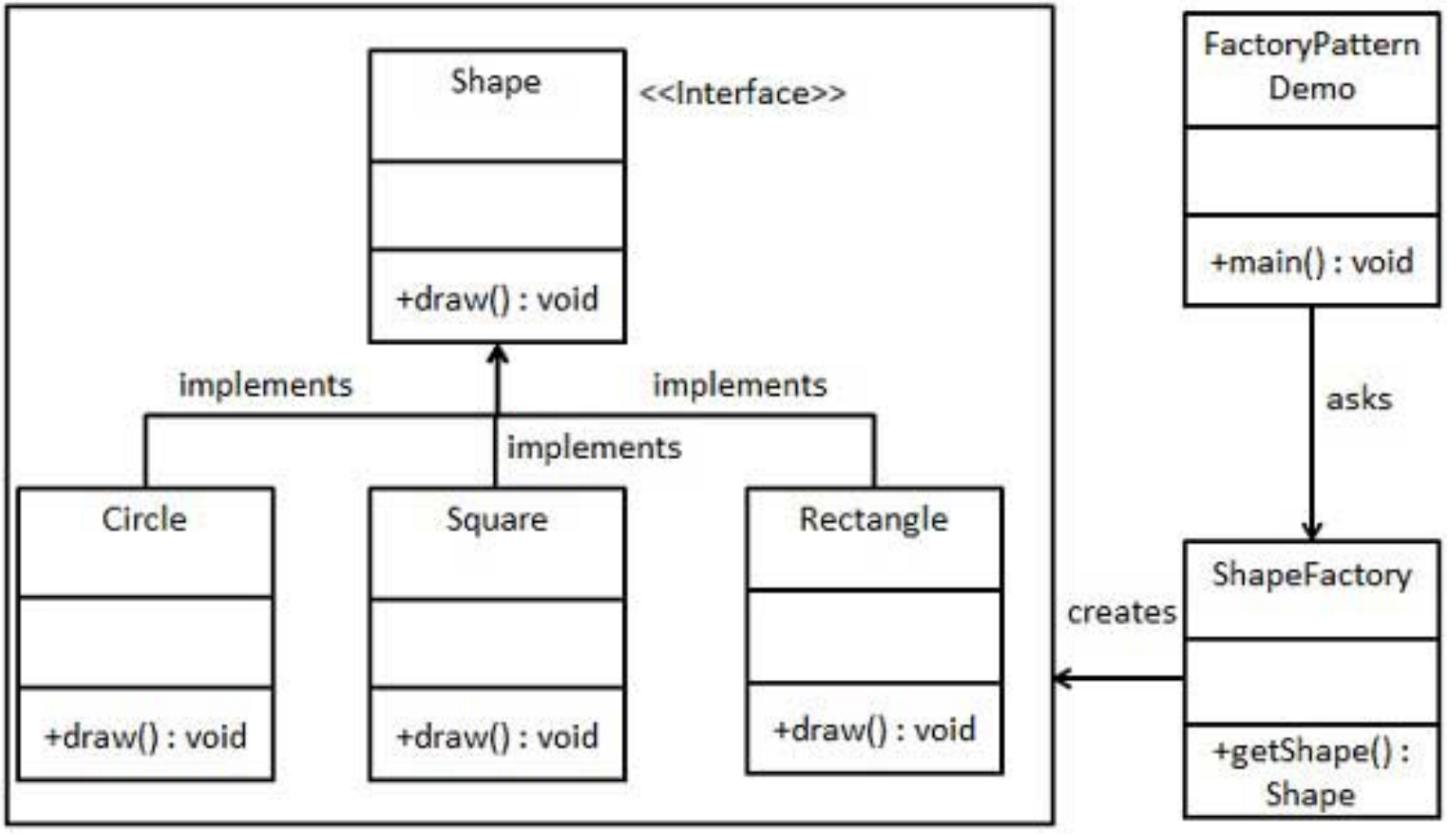
Comme toutes les API complexes en Java (et dans les langages objets en général), JSON-P utilise le *design pattern* de Factory.

Pour créer un objet de type `Foo` on ne fait pas simplement `new Foo(…)`; mais on utilise une classe `FooFactory` qui possède une méthode statique `.createFoo(…)`

Dans quel cas est-ce intéressant ?

Quand `Foo` est une interface. On ne peut pas faire `new` sur une interface. Il faut donc une méthode pour appeler le constructeur de la classe implémentant `Foo` puis qui le caste en `Foo` …

# Factory *design pattern* (exemple)



# JSON-P



L'API JSON-P se décompose en plusieurs parties (`javax.json.*`):

- Classes Java pour représenter des types JSON : objets, tableaux, nombres, chaines, booléens, `null`
- Classes pour construire les objets
- Classes pour parser du JSON depuis un flux
- Classes pour écrire du JSON dans un flux

On utilisera la classe `Json` contenant les *factories*, `JsonObjectBuilder`, `JsonArrayBuilder` pour construire des objets ou tableaux, et `.toString()` pour sérialiser.

# Exemple

```
JsonObject jobj = Json.createObjectBuilder()
                .add("nom", "Nguyễn")
                .add("prénom",  "Kim")
                .add("cours", Json.createArrayBuilder()
                                  .add("POS")
                                  .add("PAWA")
                                  .add("LCD"))
                .add("full time", true)
                .add("age", 3.6e1)
                .add("hobby", JsonValue.NULL)
                .build();

JsonReader r = Json.createReader(new FileReader("fichier.json"));
JsonObject jobjO = r.readObject(); //Si on sait que c'est un objet
JsonArray jobA = r.readArray();    //Si on sait que c'est un tableau
JsonStructure job3 = r.read();     //Sinon, tester ensuite avec instanceOf
r.close();
System.out.println(jobj);          //appelle .toString();
```