Contents

1 Les services web REST

1.1 Rappels et motivation

1.1.1 Page Web vs Service Web (WS)

1.1.1.1 Page web

1. **accès aux données** : via des pages Hypertext (*i.e. des textes avec des hyperlinks qui référencent d'autres textes*).
2. **nature** : données mélangées avec des données de présentation de la page (*e.g. HTML, CSS*).
3. **utilité** : destinée à être exploitée par des êtres humains (*e.g. Facebook et Twitter*).

1.1.1.2 Service Web

1. **définition** : un système informatique permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués.
2. **principe** : service exposé sur Internet pour un accès programmatique via une API en ligne :
   - le fournisseur de WS publie ces WS en ligne.
   - les clients utilisent/consomment les services via les données rendues disponibles en ligne.

Figure 1: Exemple d'une page web

3. **utilité** : indépendance des plateformes/langages.

1.1.1.2.1 Éléments

1. **architecture** : client-serveur.
2. **protocole de transport de messages** : HTTP.
3. **protocole d'échange de messages** :
   - **définition** : spécifier le format des messages échangés entre le client et le serveur.
   - **exemple** : SOAP.
4. **protocole de description des WS** :
   - **définition** : permet de spécifier le nom du service, le type de paramètres en entrée et le type de données en retour.
   - **exemple** : le langage WSDL avec l'API SOAP.

1.1.2 Limitations des premiers modèles WS (API SOAP)

1. HTTP est le seul moyen de transport :
   - les seules méthodes sont GET et POST.
   - chaque WS dispose d'une interface spécifique, encapsulée directement par HTTP : description en XML par WSDL.

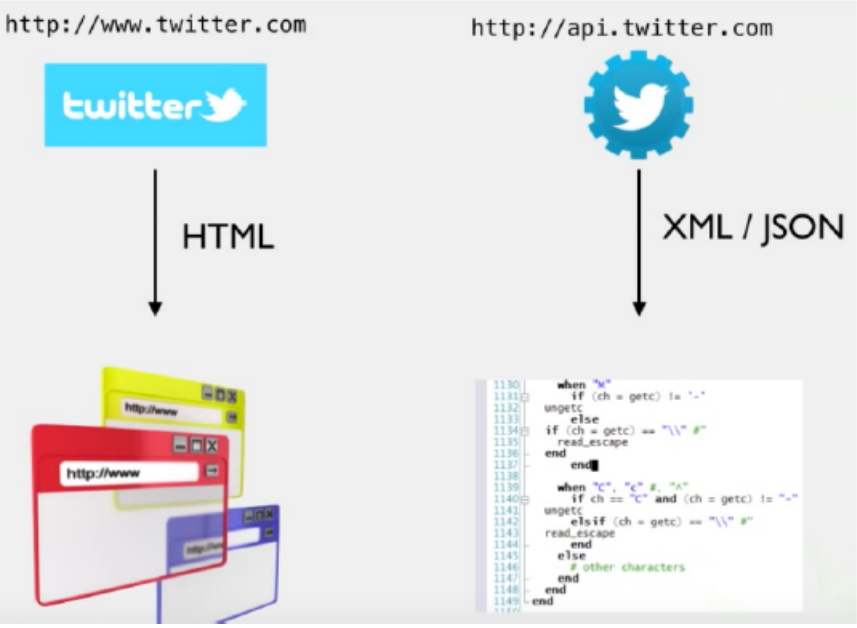

Figure 2: Exemple de comparaison entre une page web et un service web

2. architecture SOAP :
   - mise en oeuvre complexe.
   - normes volumineuses et difficiles à maîtriser.

1.2 Introduction

1.2.1 Principe

REST (**REpresentational State Transfer**) est un style architectural réseau, utilisé dans le contexte des systèmes distribués, et définissant un ensemble de guidelines :

1. **architecture** : client-serveur.
2. **transport de messages** : protocole HTTP :
   - définition de ressources identifiées par URIs.
   - utilisation de méthodes HTTP pour définir la sémantique de la communication client-serveur.
3. **échange de messages** : protocoles standardisés (*e.g. XML, JSON, MIME, ...*).
4. communication sans état (*i.e. stateless*).
5. concept non standardisé par W3C.

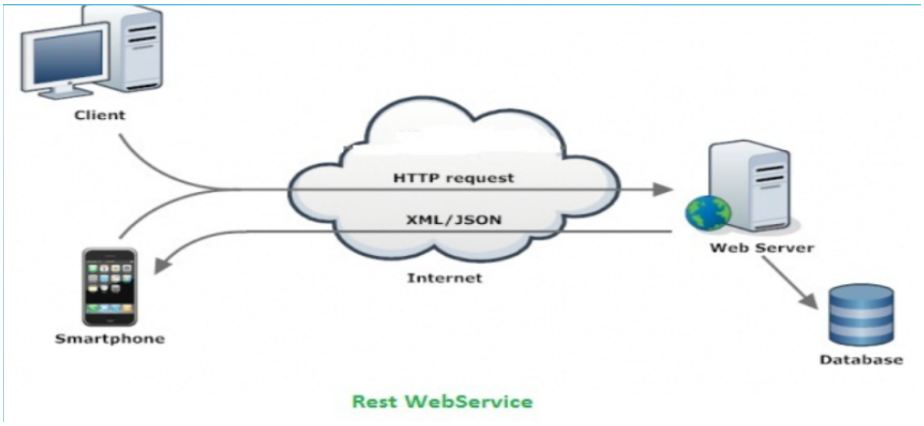

Figure 3: L'architecture REST

Remarques

1. Un(e) WS/application utilisant une API REST est dit(e) RESTful.
2. **REpresentational** → plusieurs formats de réponse sont possibles, voire même pour une même requête.

1.2.2 Origines et actualités

1. **origines** : créé par Roy Field en 2000 dans le cadre de sa thèse doctorale :
   - projet Waka.
   - Roy Field :
     1. l'un des auteurs principaux de la spécification HTTP.
     2. membre fondateur de la fondation Apache.
2. **acteurs l'utilisant** : *Google, Faceook, Amazon, Twitter, Ebay, ...*

1.2.3 REST et HTTP

1. l'inventeur de REST est l'un des auteurs principaux de HTTP.
2. pas de spécifications de standardisation pour REST (contrairement à SOAP, qui est standardisé par W3C).
3. conventions/guides pour l'utilisation de méthodes HTTP.
4. pas de précision sur les protocoles d'échange de données, et différents formats sont possibles (*e.g. XML, JSON, MIME (text), ...*).
5. pas de définition programmatique permettant de manipuler le WS, bien qu'il peut en exister une documentation consultable par les humains (*e.g. une page Web*) mais qui reste non destinée à être exploitable par un programme.

1.2.4 Avantages et Inconvénients

1.2.4.1 Avantages

1. facile à comprendre et à implémenter : des frameworks dans plusieurs langages (*e.g. Java, Python, PHP, ...*).
2. un client HTTP suffit pour consommer un WS RESTful.
3. interopérabilité des langages.
4. formats standards pour l'échange de messages (*e.g. XML, JSON, ...*) afin d'assurer la compatibilité dans le temps.
5. architecture évolutive comme conséquence de l'aspect "*stateless*" : possibilité de répartir les requêtes sur plusieurs serveurs.

1.2.4.2 Inconvénients

1. sécurité d'échange inexistante avec HTTP : il faut utiliser HTTPS + Authentification.
2. le client doit sauvegarder des données localement comme conséquence de l'aspect "stateless".

1.3 Identification/Localisation des ressources

1.3.1 Introduction

1. **définition** : les ressources d'une application web sont identifiées et localisées par des URIs.
2. **pour une page web** :
   - l'utilisateur n'a pas besoin de connaître toutes les URIs pour accéder aux pages web.
   - il a besoin uniquement de savoir l'URI de la page principale et après il pourra naviguer le site via les liens hypertext.
   - **modèle d'URIs** : action-based URI.
3. **pour une API REST** :
   - l'utilisateur a besoin d'accéder directement aux ressources via leurs URIs.
   - il a besoin de savoir la convention d'identification de ces URIs.
   - **modèle d'URIs** : resource-based URI.

1.3.2 Modèles d'URI

1.3.2.1 Action-based URI

1. **définition** :
   - une URI se focalisant sur l'action à effectuer;
   - inclue en général un verbe;

- utilise des ressources externes pour identifier la ressource sur laquelle on agit (*e.g. session state*).

2. **représentation** : `http://host/path/program?param1=value1&param2=value2&...`
3. **exemples** :
   - `http://www.lirmm.fr/~seriai~/index.php?n=Main.Software`
   - `http://weatherapp.com/weatherLookup.do?zipcode=12345`

1.3.2.2 Resource-based URI

1. **définition** :
   - une URI se focalisant sur la ressource sur laquelle on agit;
   - inclue en général une hiérarchie de noms;
   - utilise les méthodes HTTP pour définir la sémantique de l'action à effectuer (*e.g. GET, POST, PUT, DELETE, ...*).
2. **représentation** : `http://host/path/param/value`
3. **exemples** :
   - `http://weatherapp.com/zipcode/12345`
   - `http://weahterapp.com/zipcode/56789`
   - `http://weatherapp.com/countries/brazil`
   - `http://free-web-services.com/web-services/geo/weather/`
   - `http://developer.worldweatheronline.com/api/docs/`

1.3.2.2.1 Conventions de conception

1. définir une URI unique pour chaque entité accessible.
2. utiliser des noms au lieu des verbes pour désigner les ressources: *e.g.* "`document`" et "`message`" au lieu de "`getDocument`" et "`getMEssage`".
3. hiérarchiser les ressources :
   - **définition**: définir une arborescence de fichiers/dossiers dont les feuilles correspond aux URIs.
   - **exemples** :
     1. `/profiles/{profileName}` : *e.g.* `/profiles/JohnDoe`
     2. `/messages/{messageId}` : *e.g.* `/messages/1`
4. expliciter les relations entre les ressources, par exemple :
   - les commentaires et les likes sur un message donné :
     1. `/messages/{messageId}/comments/{commentId}` : `/messages/1/comments/2`
     2. `/messages/{messageId}/likes/{likeId}` : `/messages/15/likes/4`
5. distinguer les ressources uniques des collections de ressources, par exemple :
   - tous les messages : `/messages/`
   - tous les commentaires du message 15 : `/messages/15/comments`
   - tous les commentaires de tous les messages : `/messages/comments`
   - tous les commentaires : `/comments/`
   - tous les commentaires de tous les messages de tous les profiles : `/profiles/messages/comments`

6. utiliser des paramètres de filtrage et de pagination.

1.4 Requêtes/Réponses HTTP

1.4.1 Introduction

1. **description**: une requête/réponse HTTP est composée d'entêtes HTTP et d'un corps HTTP.
2. **entêtes**: des entêtes d'une requête/réponse HTTP encapsulant des méta-données pour paramétrer la communication et l'échange des messages.
3. **corps** : le corps d'une requête/réponse HTTP encapsulant les données à échanger, paramétrables par des entêtes HTTP, et qui peut éventuellement être vide, selon la méthode HTTP utilisée.

1.4.2 Entêtes d'une requête HTTP

1. **entrées standards** : méthode HTTP, URI de la ressource, HTTP et sa version.
2. **entrées de paramétrage des messages** :
   - `Content-Type` :
     1. **description** : type de données du contenu du message envoyé (*e.g. XML, JSON, MIME, ...*).
     2. **exemple** : `Content-Type: application/json`.
   - `Content-Length` : longueur du message envoyé en octets.
   - `Date` :
     1. **description** : date de la requête du client.
     2. **format** : `jour_semaine, jour_mois mois année heures:minutes:secondes GMT`
     3. **exemple** : `Date: Tue, 19 Jan 2016 18:15:41 GMT`
   - `Referer` :
     1. **description** : la localisation précédente du client.
     2. **exemple** : `Referer: https://www.google.fr/`
   - `User-Agent` :
     1. **description** : l'application utilisée par le client pour envoyer la requête.
     2. **exemple** : `User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5)`
   - `Cookie` :
     1. **description** : un cookie sauvegardé par le client.
     2. **exemple** : `Cookie: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX` (`chaîne alphanumérique`).
3. **entrées de** *content negotation* : choisir le format des données retournées par le serveur :
   - `Accept` :

        1. **description** : le(s) format(s) du contenu du message reçu accepté(s) par le client.
        2. **exemple** : `Accept: text/html, application/xhtml+xml, application/xml;q=0.9`
    - `Accept-Language` :
        1. **description** : le(s) language(s) accepté(s) par le client.
        2. **exemple** : `Accept-Language: en-us,en;q=0.8`
4. ...

1.4.3 Entêtes d'une réponse

1. **entrées standards** : HTTP et sa version, HTTP status code, HTTP status message.
2. **entrées de métadonnées** :
    - `Content-Type` : type de données du contenu du message reçu.
    - `Content-Length` : longueur du message reçu en octets.
    - `Date` : date de la réponse du serveur.
3. ...

1.4.4 HTTP status codes/messages

1. **définition** : le code et le message de retour correspondant d'un serveur HTTP.
2. **rôle** : indiquer l'état d'exécution requête HTTP envoyée.

1.4.4.1 Catégories

Category	Description	Examples (`code: message`)
`1XX`	code informationnel	–
`2XX`	code de succès	`200: ok`; `201: created`; `204: no content`
`3XX`	code de redirection	`301: moved permanently`; `302: found`; `304: not modified`; `307: temporary redirect`

Category	Description	Examples (`code: message`)
`4XX`	code erreur côté client	`400: bad request`; `401: unauthorized`; `403: forbidden`; `404: not found`; `415: unsupported media type`
`5XX`	code erreur côté serveur	`500: internal server error`

1.5 Méthodes HTTP

1.5.1 Introduction

1. **définition** : une méthode HTTP est utilisée pour spécifier la sémantique d'une requête/réponse HTTP.
2. **les méthodes les plus communes** :
   - `GET`, `POST`, `PUT`, `DELETE`.
   - correspondent aux opérations **CRUD** (**Create Read Update Delete**).
3. **des méthodes moins utilisées** : `HEAD`, `OPTIONS`, ...
4. **remarque** : un WS bien conçu avec une API REST, doit se baser sur les méthodes HTTP principales: GET, POST, PUT, DELETE.

1.5.2 Types

1.5.2.1 La méthode `GET`

1. **définition** : récupérer la représentation d'une ressource/collection de ressources depuis un serveur HTTP.
2. **propriété** : méthode de lecture.
3. **exemples** :
   - `/messages/20` : récupérer le message 20.
   - `/messages/20/comments` : récupérer tous les commentaires associés au message 20.
   - `/messages` : récupérer tous les messages.

1.5.2.2 La méthode `POST`

1. **définition** : créer une ressource/collection de ressources sans spécifier son URI.
2. **propriété** : méthode d'écriture.
3. **remarques**
   - l'URI indiquée pour l'action sera celle d'une ressource parente (*e.g. un dossier*) dans l'hiérarchie des ressources.
   - le serveur décide quelle URI à attribuer à la ressource créée.
4. **exemples** :
   - `/messages` : créer un nouveau message.
   - `/messages/20/comments` : créer un nouveau commentaire associé au message 20.

1.5.2.3 La méthode `PUT`

1. **définition** : créer/modifier une ressource/collection de ressources (existante) en spécifiant son URI.
2. **propriété** : méthode d'écriture.
3. **exemples** :
   - `/messages/20` : remplacer/créer le message 20.
   - `/messages/20/comments/10` : remplacer/créer le commentaire 10 associé au message 20.
   - `/messages/20/comments` : remplacer/créer tous les commentaires associé au message 20.

1.5.2.4 La méthode `DELETE`

1. **définition** : supprimer une ressource/collection de ressources sur un serveur HTTP.
2. **propriété** :
   - méthode d'écriture.
   - ne permet pas d'assurer la suppression du contenu souhaité.
3. **exemples** :
   - `messages/20` : supprimer le message 20.
   - `messages/20/comments/10` : supprimer le commentaire 10 associé au message 20.
   - `messages/20/comments` : supprimer tous les commentaires associés au message 20.

1.5.2.5 La méthode `HEAD`

1. **définition** : envoyer des informations sur une ressource.

1.5.3 Idempotence

1. **définition** : une méthode dont l'invocation plusieurs fois dans des requêtes identiques répétées produit le même résultat (i.e. sans effet supplémentaire du côté serveur après chaque invocation) est dite idempotente
2. les méthodes `GET`, `PUT` et `DELETE` sont idempotentes, alors que la méthode `POST` ne l'est pas.
3. **exemples** :

```
GET /messages/20 : récupérer le message 20.
GET /messages/20 : récupérer le message 20.
$\dots$
PUT /messages/20 : remplacer/créer le message 20.
PUT /messages/20 : remplacer/créer le message 20.
$\dots$
DELETE /messages/20 : supprimer le message 20.
DELETE /messages/20 : supprimer le message 20.
$\dots$
POST /messages : créer un nouveau message (e.g. messages/21).
POST /messages : créer un nouveau message (e.g. messages/22).
POST /messages : créer un nouveau message (e.g. messages/23).
$\dots$
```

1.6 Processus de création d'une API REST

1. **définition des ressources manipulées et concevoir leurs URIs** : *ressource unique ou collection de ressources ?*
2. **codage de la représentation des ressources** :
   - *quel format à utiliser ?*
   - *quels attributs possède une ressource ?*
3. **spécification de la sémantique des messages échangés** : *quelles méthodes HTTP pour quelles ressources ?*
4. **spécification de la réponse REST** :
   - *quel type de données pour une réponse ?*
   - *quel code de retour pour une réponse ?*
5. **remarque** : une ressource n'est pas forcément accessible par tous les messages du protocole de transport : *e.g. une ressource accessible en lecture uniquement n'est accessible que par la méthode HTTP GET*

1.7 Techniques et outils supplémentaires

1.7.1 HATEOAS (Hypermedia As The Engine Of Application State)

1.7.1.1 Introduction

1. **motivation** : dans le contexte d'une **API REST**, le client doit connaître toutes les URIs pour accéder aux ressources.
2. **définition** : **HATEOAS** est une technique permettant d'adopter le mécanisme de navigation d'une page web pour pallier à l'absence des URIs nécessaires à la navigation (*i.e. connaître la page racine et naviguer en utilisant des hyperlinks*).
3. **mécanisme** : lors de l'envoi d'une réponse HTTP à une requête, inclure dans son corps les URIs des ressources pertinentes.

1.7.2 Exemple en JSON

```
{
  "id": "20",
  "message": "hello world",
  "date": "24Apr2016",
  "author": "doe",
  "links": [
    {
      "href": "/messages/1",
      "rel": "self"
    },
    {
    "comments-href": "api/messages/20/comments",
    "rel": "comments"
    },
    {
      "likes-href": "api/messages/20/likes",
      "rel": "likes"
    },
    {
      "shares-href": "api/messages/20/shares",
      "rel": "shares"
    }
  ]
}
```

1.7.3 Richardson Maturity Model

1.7.3.1 Introduction

1. **définition** : une classification des applications REST en fonction de leur conformité aux conventions du style REST.
2. **classes** :
   - `niveau 0` : application non RESTful.
   - `niveau 1` : utilisation des URIs.

- `niveau 2` : utilisation des méthodes HTTP.
- `niveau 3` : **HATEOAS**.

1.7.3.2 Exemple

Une API web utilisant le style architectural de réseaux `PlainXML`, qui encapsule toutes les ressources dans une même URI (*e.g. API SOAP*), est une application classée au niveau 0 (*non RESTful*) dans le modèle de maturité de Richardson.

1.7.4 WADL (Web Application Description Language)

1.7.4.1 Introduction

1. **définition** : un langage de description des WS RESTful au format XML.
2. **origines** : initié par SUN micorsystems et standardisé par W3C.
3. **principe** :
   - fournir des informations descriptives d'un WS RESTful permettant de construire des applications clientes pouvant l'exploiter.
   - décrire des éléments à partir de leur type (*ressource, verbe, paramètre, type de requêtes, réponse*).
   - interagir de manière dynamique avec les applications REST.
4. **caractéristique** : moins exploité que WSDL avec les WS SOAP.

1.7.4.2 Exemple

```
<!--../source/rest/xml/snippets/wadl_example.xml-->

?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <doc xmlns:jersey="http://jersey.java.net"
        jersey:generatedBy="Jersey: 2.0 2013-05-03 14:50:15" />
    <grammars />
    <resources base="http://localhost:8080/Bibliotheque/webresources/">
        <resource path="category">
            <method id="test" name="GET">
                <response>
                    <representation mediaType="application/xml"/>
                    <representation mediaType="application/json"/>
                </response>
            </method>
            <method id="apply" name="OPTIONS">
                <request>
                    <representation mediaType="*/*"/>
                </request>
                <response>
```

```
                    <representation mediaType="application/vnd.sun.wadl+xml"/>
                </response>
            </method>
            <method id="apply" name="OPTIONS">
                <request>
                    <representation mediaType="*/*"/>
                </request>
                <response>
                    <representation mediaType="text/plain"/>
                </response>
            </method>
            <method id="apply" name="OPTIONS">
                <request>
                    <representation mediaType="*/*"/>
                </request>
                <response>
                    <representation mediaType="*/*"/>
                </response>
            </method>
        </resource>
        <resource path="application.wadl">
            <method id="getWadl" name="GET">
                <response>
                    <representation mediaType="application/vnd.sun.wadl+xml"/>
                    <representation mediaType="application/xml"/>
                </response>
            </method>
            <method id="apply" name="OPTIONS">
                <request>
                    <representation mediaType="*/*"/>
                </request>
                <response>
                    <representation mediaType="text/plain"/>
                </response>
            </method>
            <method id="apply" name="OPTIONS">
                <request>
                    <representation mediaType="*/*"/>
                </request>
                <response>
                    <representation mediaType="*/*"/>
                </response>
            </method>
        </resource>
    </resources>
</application>
```

1.8 API SOAP vs. API REST

1.8.1 SOAP

1. **avantages** :
   - standardisé W3C.
   - interopérabilité des langages.
   - plusieurs protocoles de transport possibles (HTTP, SMTP, FTP, ...).
   - sécurité avec WS-Security.
2. **limitations** :
   - performance réduite causée par l'enveloppe SOAP supplémentaire.
   - complexité et lourdeur d'implémentation.

1.8.2 REST

1. **avantages** :
   - basé données.
   - implémentation simple.
   - lisibilité par les humains.
   - repose sur les principes Web.
   - formats de données échangées standardisés : *e.g. XML, JSON, MIME, ...*
   - évolutivité en raison de l'aspect "*stateless*".
   - pas d'infrastructure dédiée.
2. **limitations** :
   - sécurité restreinte par l'emploi des méthodes HTTP.
   - pas de conservation de l'état des échanges.

2 Services web REST en Java, workflow, astuces, et bonnes pratiques

2.1 Quelques notions de base

2.1.1 Couplage, découplage

1. le **couplage** entre deux classes `A` et `B` désigne le degré de dépendance entre elles, tel que toute modification à `A` entraîne une modification à `B` et/ou inversement.
2. une conception à **couplage fort** d'un système entraîne des limitations sur la réutilisation, l'extensibilité, la testabilité, la maintenance, et l'évolution des entités du système. Le but ainsi est d'assurer une conception à **couplage faible** afin d'alléger ces limitations.

3. **découpler** les entités d'un système consiste à en réduire les dépendances, ce qui peut se faire en se basant sur des **prinicipes de conception** (*design principles*) divers, concrétisés à leurs tours par des **mécanismes d'implémentation** divers (*e.g., des* ***patrons de conception*** *(design patterns)*).

2.1.2 Principe de conception (*Design Principle*) vs. Patron de conception (*Design Pattern*)

1. **principe de conception**:
   - une guide de haut niveau pour la conception d'applications logicielles de haute qualité.
   - ne fournit aucun détail d'implémentation.
   - ne dépend pas d'un langage de programmation.
   - **exemple** : le **Single Responsibility Principle** (*SRP*) indiquant qu'une classe ne doit avoir qu'une seule raison pour changer. Autrement dit, une classe d'un système doit uniquement se concentrer sur sa responsabilité métier principale.
2. **patron de conception**:
   - une solution d'implémentation de bas niveau pour la résolution d'un problème survenu régulièrement lors de la conception d'une application logicielle.
   - basée sur l'expérience des développeurs → testée avec plusieurs cas d'usage et assez fiable.
   - **example**: le patron de conception **Singleton** se présentant comme une solution du problème : *Comment avoir une classe ne pouvant être instanciée qu'une seule fois ?*

2.1.3 Les principes de conception SOLID

Parmi les principes de conception les plus connus et adoptés dans le monde du génie logiciel pour la réalisation d'un système orienté-objet faiblement couplé sont les **principes SOLID** introduits par Robert Martin (*Uncle Bob*) :

1. **S**ingle Responsibility Principle (*SRP*) : une classe ne doit avoir qu'une seule raison pour changer. Autrement dit, une classe d'un système doit uniquement se concentrer sur sa responsabilité métier principale.
2. **O**pen/Closed Principle (*OCP*) : une classe doit rester ouverte pour toute extension mais fermée pour toute modification. Autrement dit, le code source d'une classe doit rester intact et toute modification ultérieure nécessaire doit être introduite par le biais d'un mécanisme d'extension (*héritage, composition*).
3. **L**iskov Substitution Principle (*LSP*) : toute instance d'une classe dérivée peut être remplacée par une instance de sa classe parente sans introduire des incohérences comportementales au niveau du système. Autrement dit,

pour un programme `P`, un type `S` est un **sous-type** d'un autre type `T`, si et seulement si tout objet `O1` de type `S` peut être remplacé par un autre objet `O2` de type `T` sans changer le comportement de `P`.

4. **I**nterface Segregation Principle (*ISP*) : opter pour la création de petites interfaces spécifiques et moins générales pour offrir plus de flexibilité et configurabilité aux classes les implémentant. Autrement dit, il s'agit de la projection du principe **SRP** dans le monde des interfaces, afin d'assurer que chaque interface ne s'occupe d'offrir qu'une seule responsabilité métier bien définie pour chaque classe l'implémentant.
5. **D**ependency Inversion Principle (*DIP*) :
   - découpler les **modules haut-niveau** d'un système (*haut niveau dans ce contexte désignent les modules qui dépendent d'autres modules*) de **ses modules bas-niveau** (*bas niveau dans ce contexte désignent les modules dont d'autres modules en dépendent*) et les rendre dépendants d'**abstractions** (*descriptions générales telles que les interfaces*) au lieu de **concrétisations** (*implémentations spécifiques par des classes*).
   - les abstractions ne doivent pas dépendre des concrétisations, mais plutôt l'inverse.
   - autrement dit, il faut supprimer les dépendances qui existent directement entre les modules haut niveau et les modules bas niveau et les remplacer par des dépendances intermédiaires désignant des abstractions au lieu de concrétisations.

Pour plus de détails consulter la section `Références`.

2.1.4 Aperçu sur les frameworks

2.1.4.1 Frameworks

1. **définition** :
   - un **framework** est une application logicielle partielle :
     1. intégrant les connaissances d'un domaine ;
     2. dédiée à la réalisation rapide de nouvelles applications du domaine visé ;
     3. dotée d'un coeur (*core*) générique, extensible et adaptable.
   - un ensemble de classes coopérant dans le cadre d'un schéma permettant une conception réutilisable par un type spécifique de logiciels.
2. **utilité** : fournir une guide architecturale pour la création d'applications en partionnant sa conception en un ensemble de classes abstraites et en définissant leurs responsabilités et collaborations.
3. **principe d'usage** : un développeur personnalise le framework par héritage et/ou composition de ses classes pour la mise en oeuvre d'une application particulière.

2.1.4.2 Framework vs. bibliothèque

1. **similarité** : les bibliothèques et frameworks à objets sont basés sur les mêmes mécanismes de paramétrage (*i.e. par spécialisation/composition*).
2. **différences** :
   - *une bibliothèque s'utilise alors qu'un framework s'étend.*
   - le code d'une nouvelle application :
     1. *utilisant une bibliothèque* : invoque le code de la bibliothèque.
     2. *utilisant un framework* : est invoqué par le code du framework.

2.1.4.3 Inversion de contrôle (*Inversion of Control* (*IoC*)) ou le principe de Hollywood

1. l'inversion de contrôle est un **principe de design** qui consiste en l'inversion du flot de contrôle tel qu'il est défini traditionnellement dans le paradigme de la programmation procédurale.
2. dans le paradigme procédural, le flot de contrôle est initié par les parties de code représentant l'application développée par le client, et qui invoquent des bouts de code réutilisables et répertoriés au sein de librairies logicielles.
3. avec l'IoC, le flot de contrôle est initié par le coeur d'un framework qui invoque/*callback* les parties de code représentant la nouvelle application développée par le client dans certains endroits prédéfinis et documentés, nommés **points d'extension**, **points de paramétrage** ou *hot spots*.
4. cette invocation en callback par le framework est à l'origine du nom du principe de Hollywood : *Don't call us, we'll call you.*
5. l'IoC peut être implémentée par plusieurs patrons de conceptions, notamment par **l'injection de dépendance** (*Dependency Injection* (*DI*)).

2.1.5 Injection de dépendance (*Dependency Injection* (*DI*))

2.1.5.1 Introduction

1. **définition** : un patron de conception implémentant le principe de conception **IoC** pour inverser la création et la gestion du cycle de vie des instances des dépendances d'une classe.
2. **besoin** : découpler l'usage des dépendances de leur création et de leur gestion → les dépendances peuvent être créées/gérées de manières différentes.
3. **motivation** : dans le monde des frameworks :
   - l'IoC nécessite que le contrôle d'une application sera passé à des extensions (*s'il y en a*) à travers des points d'extension.
   - indiquer au framework à qui passer le contrôle est réalisé par une injection de dépendence.
   - de ce point de vue, une injection de dépendance est une association d'une extension à un point d'extension.

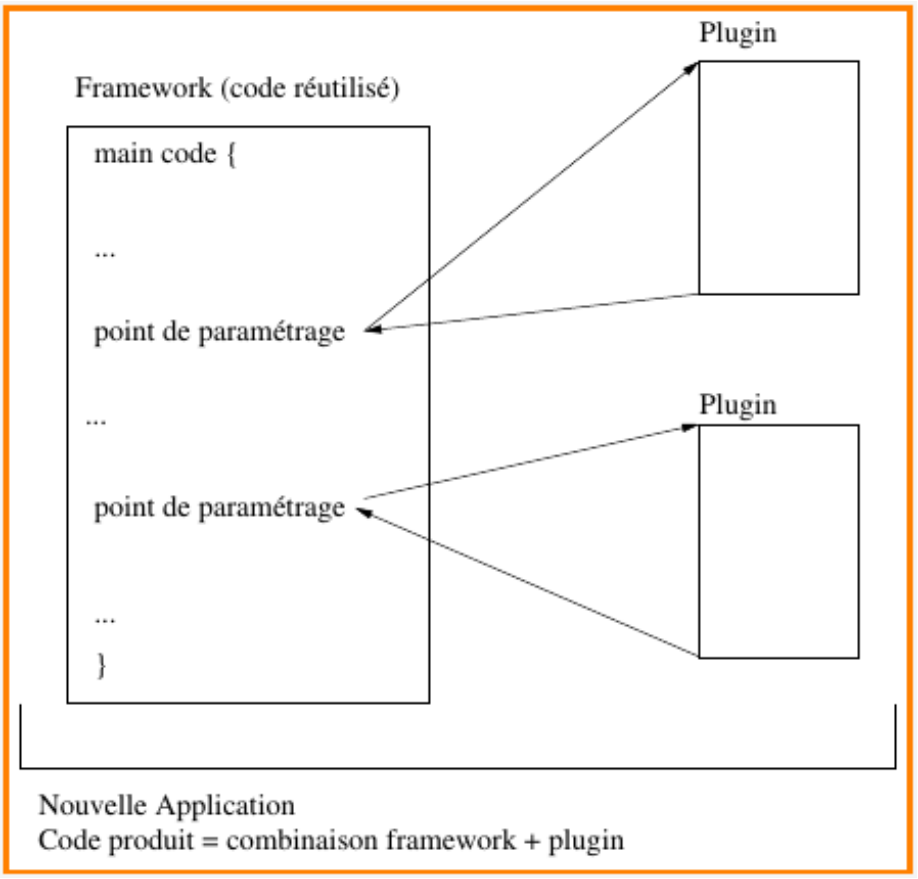

Figure 4: Inversion de contrôle

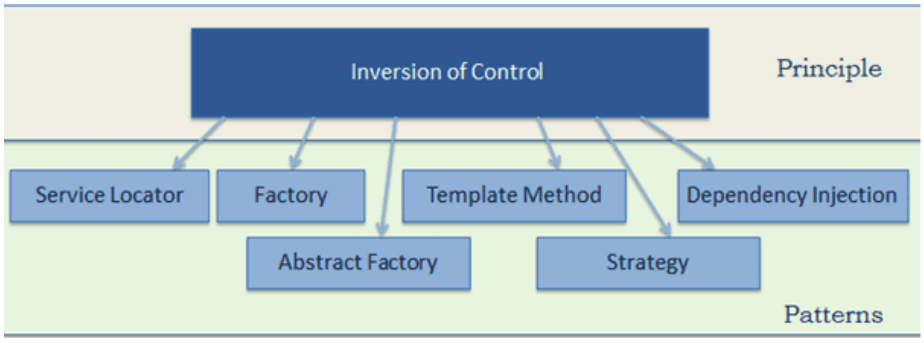

Figure 5: Les patrons de conception qui peuvent implémenter l'inversion de contrôle

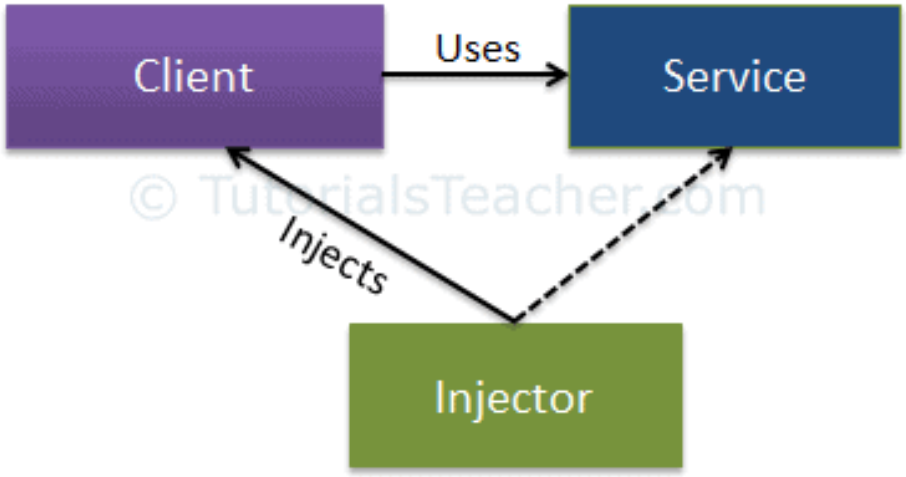

Figure 6: Le patron de conception Dependency Injection

2.1.5.2 Participants

1. `Client`: une classe dépendante d'une autre classe `Service`.
2. `Service`: une classe de dépendance fournissant un service à sa classe dépendante `Client`.
3. `Injector`: une classe fournissant des instances de `Service` à `Client`.

2.1.5.3 Types

1. **injection de dépendance par constructeur** : `Injector` fournit une instance à `Client` via un constructeur de celle-ci.
2. **injection de dépendance par une méthode setter** : `Injector` fournit une instance à `Client` via une méthode setter de celle-ci.
3. **injection de dépendance par une interface** :
   - `Client` implémente une interface `IInjectable` déclarant la/les méthode(s) fournissant une/des instance(s) de `Service`.
   - `Injector` utilise les méthodes de `IInjectable` afin de fournir la/les instance(s) de `Service` à `Client`.

2.1.5.4 Avantages

1. fournit un moyen facile pour la communication entre les composants d'une application.
2. réduit le couplage entre les classes et facilite ainsi son extensibilité, testabilité, maintenabilité et évolution.
3. réduit le code non métier (*i.e., boilerplate*) de l'application.

2.1.5.5 Exemple

```
// ../source/rest/java/dependency_injection/ICustomerDataAccess.java

ackage others.dependency_injection;

public interface ICustomerDataAccess {
    /* METHODS */
    String getCustomerName(int id);
}
```

```
// ../source/rest/java/dependency_injection/CustomerDataAccess.java

ackage others.dependency_injection;

public class CustomerDataAccess implements ICustomerDataAccess {

    /* CONSTRUCTOR */
    public CustomerDataAccess() {

    }

    /* METHODS */
    @Override
    public String getCustomerName(int id) {
        return String.format("Name of customer with ID %d", id);
    }
}
```

```
// ../source/rest/java/dependency_injection/CustomerBusinessLogic.java

ackage others.dependency_injection;

public class CustomerBusinessLogic {
    /* ATTRIBUTES */
    protected ICustomerDataAccess dataAccess;

    /* CONSTRUCTORS */
    public CustomerBusinessLogic() {
        dataAccess = new CustomerDataAccess();
    }

    // Interface for Dependency Injection by Constructor
    public CustomerBusinessLogic(ICustomerDataAccess dataAccess) {
        System.out.println("Constructor Dependency Injection: ");
        this.dataAccess = dataAccess;
    }
```

```
    /* METHODS */
    public String getCustomerName(int id) {
        return dataAccess.getCustomerName(id);
    }

    // Interface for Dependency Injection by Setter Method
    public void setCustomerDataAccess(ICustomerDataAccess dataAccess) {
        System.out.println("Setter Dependency Injection: ");
        this.dataAccess = dataAccess;
    }
}
```

```
// ../source/rest/java/dependency_injection/InjectableWithCustomerDataAccess.java

ackage others.dependency_injection;

public interface InjectableWithCustomerDataAccess {
    void inject(ICustomerDataAccess dataAccess);
}
```

```
// ../source/rest/java/dependency_injection/InterfaceInjectableCustomerBusinessLogic.java

ackage others.dependency_injection;

public class InterfaceInjectableCustomerBusinessLogic extends
    CustomerBusinessLogic implements InjectableWithCustomerDataAccess{

    @Override
    public void inject(ICustomerDataAccess dataAccess) {
        System.out.println("Interface Dependency Injection: ");
        this.dataAccess = dataAccess;
    }
}
```

```
// ../source/rest/java/dependency_injection/CustomerService.java

ackage others.dependency_injection;

public class CustomerService {
    /* ATTRIBUTES */
    private CustomerBusinessLogic businessLogic;

    /* CONSTRUCTORS */
    public CustomerService() {

    }
```

```java
    /* METHODS */
    // Dependency Injection by constructor
    public static CustomerService serveWithConstructorInjection(
            ICustomerDataAccess dataAccess) {
        CustomerService service = new CustomerService();
        service.businessLogic = new CustomerBusinessLogic(dataAccess);

        return service;
    }

    // Dependency Injection by setter
    public static CustomerService serveWithSetterInjection(
            ICustomerDataAccess dataAccess) {
        CustomerService service = new CustomerService();
        service.businessLogic = new CustomerBusinessLogic();
        service.businessLogic.setCustomerDataAccess(dataAccess);

        return service;
    }

    // Dependency Injection by injectable interface
    public static CustomerService serveWithInterfaceInjection(
            ICustomerDataAccess dataAccess) {
        CustomerService service = new CustomerService();
        service.businessLogic = new InterfaceInjectableCustomerBusinessLogic();
        ((InterfaceInjectableCustomerBusinessLogic) service.businessLogic)
            .inject(dataAccess);

        return service;
    }

    public String getCustomerName(int id) {
        return businessLogic.getCustomerName(id);
    }
}
// ../source/rest/java/dependency_injection/Main.java

ackage others.dependency_injection;

public class Main {

    public static void main(String[] args) {
        ICustomerDataAccess dataAccess = new CustomerDataAccess();
        CustomerService service = CustomerService
```

```
                .serveWithConstructorInjection(dataAccess);
        System.out.println(service.getCustomerName(1));
        System.out.println();
        // Constructor Dependency Injection:
        // Name of customer with ID 1

        service = CustomerService
                .serveWithSetterInjection(dataAccess);
        System.out.println(service.getCustomerName(1));
        System.out.println();
        // Setter Dependency Injection:
        // Name of customer with ID 1

        service = CustomerService
                .serveWithInterfaceInjection(dataAccess);
        System.out.println(service.getCustomerName(1));
        // Interface Dependency Injection:
        // Name of customer with ID 1
    }
}
```

2.1.6 Conteneurs d'inversion de contrôle

1. **définition**: un **conteneur d'inversion de contrôle** (*IoC container*) est un framework utilisé pour la gestion automatique de la création, le cycle de vie, et l'injection des dépendances lors de l'exécution d'une application, libérant ainsi les développeurs et leur permettant de se concentrer sur la logique métier du système développé.
2. **processus** :
   - enregistrement d'un map entre un type rencontré et ses dépendances correspondantes.
   - pour chaque type rencontré ayant des dépendances, création des objets des dépendances correspondantes.
   - injection des objets lors de l'exécution de l'application par un mécanisme d'injection de dépendance (*par constructeur, par une méthode setter, ou par des méthodes d'injection d'une interface*).
   - gestion du cycle de vie des objets de dépendances créés et libération de leur espace mémoire alloué convenablement.
3. **examples**: le framework Java `Spring`, le framework .NET `Unity`, ...

2.1.7 Aperçu d'une architecture N-Tiers

Une **architecture N-Tiers** d'une application permet de la structurer en plusieurs couches, notamment :

1. la **couche des données** : les entités constituant le modèle de données de l'application et leur persistance sur un support comme une base de données.
2. la **couche d'accès aux données** : pour faire le mapping entre le monde des objets et le monde des données.
3. la **couche métier** : pour réaliser la logique métier de l'application sur les objets récupérés depuis la couche d'accès aux données.
4. la **couche de présentation** : pour faire le mapping entre la couche métier et l'interface d'interaction avec le client.
5. l'**interface d'interaction avec le client** : une CLI/GUI permettant de mapper les requêtes de l'utilisateur à la couche de présentation de l'application.

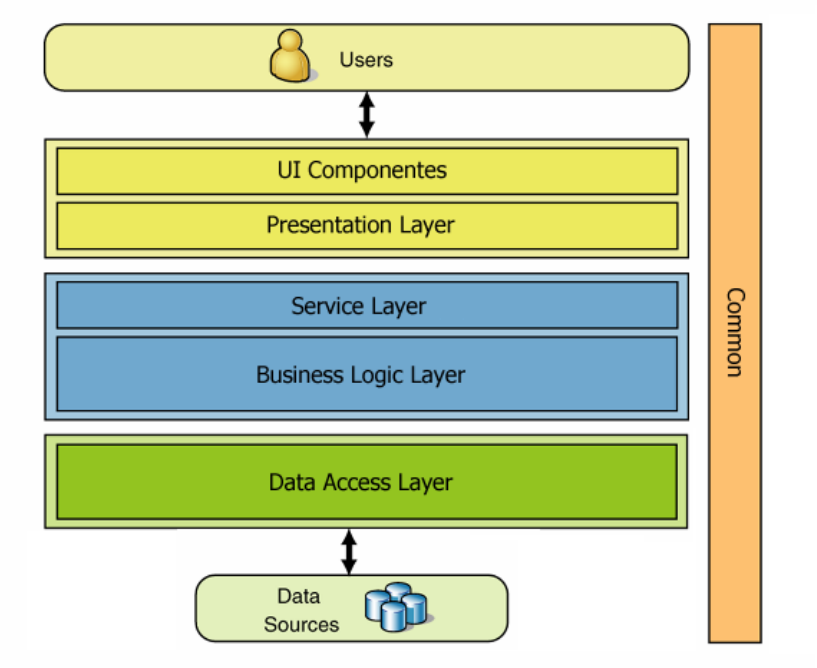

Figure 7: Architecture N-Tiers

2.2 Aperçu de Spring

2.2.1 Introduction

1. Spring est un framework open-source et modulaire pour la création d'applications N-Tier Java standards, web, et JEE (*applications d'entreprise*).

2. créé en 2003.

2.2.2 Features

1. Spring permet de développer des applications Java en utilisant des **POJOs** (**Plain Old Java Objects**) modélisant des **servlets** (*encapsulant la logique métier*) et des **pages JSP** (*encapsulant la présentation*).
2. Spring est organisé d'une manière modulaire permettant à une application d'utiliser uniquement les modules dont elle a besoin.
3. Spring ne réinvente pas tout *from scratch*, mais utilise plutôt un ensemble de technologies Java existantes, telles que les frameworks **ORM** (**Object-Relation Mapping**), de logging, JEE, ...
4. Spring fournit une modélisation **MVC** (**Model-View-Controller**) d'une application avec Spring MVC.
5. Spring fournit un conteneur IoC permettant de réutiliser des composants faiblement couplés entre plusieurs applications. La mise en place du système se fait via le mécanisme d'**injection de dépendances** au niveau : (1) des constructeurs, ou (2) des méthodes setters.
6. Spring fournit une interface pour la gestion des transactions avec les base de données.

2.2.3 Architecture et aperçu des modules

Le framework Spring est constitué d'environ 20 modules dont chacun peut être utilisé par une applications Spring selon ses besoin.

Certains modules sont groupés en conteneurs tels que :

1. le conteneur `Core` contenant les modules `Core`, `Beans`, `Context`, et `Expression Language`, et constituant les parties fondamentales du framework telles que le conteneur IoC et les mécanismes d'injection de dépendances.
2. le conteneur `Data Access/Integration` contient les modules liés à la couche d'accès aux données, incluant `JDBC`, `ORM`, `OXM`, `JMS` et `Transaction`.
3. le conteneur `Web` consistant de modules dédiés au développement des applications web, incluant `Web`, `Web-MVC`, `Web-Socket`, et `Web-Portlet`.

D'autres modules aussi non conteneurisés sont proposés par Spring, incluant les modules `AOP`, `Aspects`, `Instrumentation`, `Web` et `Test`.

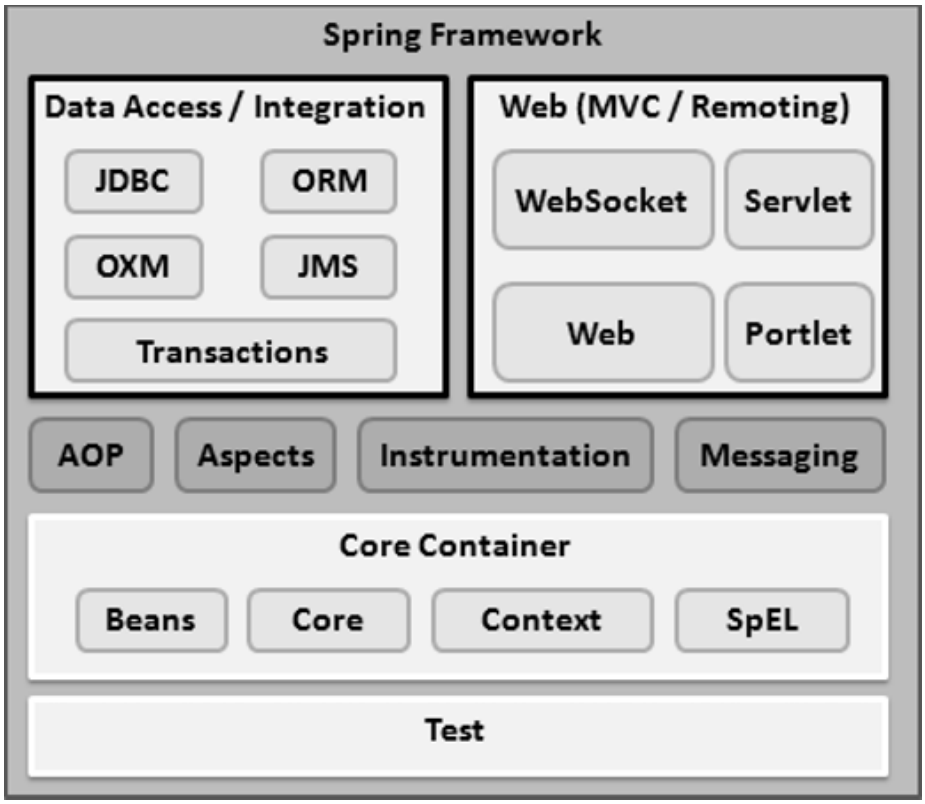

Figure 8: Architecture de Spring

2.3 Aperçu de Spring Boot

2.3.1 Introduction

1. Spring Boot est un module du framework Spring désignant un framework open-source qui simplifie l'usage de Spring pour le développement flexible et rapide d'applications *standalone* en Java, et qui peuvent éventuellement être hébergées dans le cloud.
2. la simplification est assurée par l'usage de quelques fonctionnalités telles que : la gestion automatique des dépendances, la fourniture de serveurs HTTP prêts à l'usage, la configuration automatique des composants de l'application, la gestion des endpoints, et l'usage d'outils tels que Spring CLI.

2.3.2 Features

1. `Spring CLI` : permet l'usage du langage Groovy pour la définition et la mise en place rapide d'applications Spring Boot en cachant aux développeurs tout le code non métier ou *boilerplate* (*configuration, infrastructurel, ...*).
2. `Starter Dependency`: permet l'ajout et la gestion automatique des dépendances lors de la définition du projet afin d'améliorer la productivité des développeurs.
3. `Spring Initializer`: permet la création d'une application web avec une structure interne.
4. `Auto-configuration`: permet la configuration automatique par défaut des entités utilisées dans un projet Spring en utilisant la conteneur d'injection de dépendances de Spring.
5. `Spring Actuator`: fournit une API permettant de monitorer la performance et l'état d'une application Spring Boot en temps réel.
6. `Logging and Security`: fournit des mécanismes de sécurité et de logging afin de sécuriser le système et en générer des traces d'exécution pour son débogage, compréhension, maintenance, évolution, ...

2.4 Setup

2.4.1 Pré-requis

1. JRE et outils Java : JDK $\geq$ 1.8.

2.4.2 Installation de Spring Boot en Eclipse

1. `Help` $\rightarrow$ `Eclipse Marketplace...`

2. rechercher et installer `Spring Tools 4 (aka Spring Tool Suite 4)`.
3. redémarrer Eclipe après l'installation.

2.5 Créer un service web REST avec Spring Boot

2.5.1 Workflow

1. Créer votre projet Spring Boot avec `Spring Starter Project` et choisissez les dépendances (*e.g., Spring Web, Spring Data, H2, ...*).
2. Créer les classes de votre modèle de données (*i.e., les ressources à exposer*).
3. Créer les repositories correspondant à vos ressources.
4. Créer les services web nécessaires pour exploiter vos ressources.
5. Ajouter les méthodes HTTP pour les opérations CRUD pour chacun de vos services web :
   - Spécifier les URIs pour consommer chaque opération de vos services web.
   - Spécifier les types de données pour les entrées/retours de vos opérations
   - Spécifier les codes HTTP pour les réponses retournées par vos opérations.
6. Configurer votre application Spring Boot pour permettre au mécanisme de l'injection de dépendances de trouver vos entités, composants, configurations, ...
7. Configurer votre application dans `application.properties` afin de changer le numéro de port de votre serveur Tomcat, afin de pouvoir accéder en console à votre base de données, ...
8. Tester votre service web REST avec des applications utilisant des clients RESTful (*e.g., cURL, Postman, toute autre application utilisant un client RESTful*).

2.5.2 Exemple d'un service web REST exposant des informations sur des employés

Supposons que l'on dispose d'un ensemble d'employés ayant chacun un **identifiant** (*un entier*), un **nom** (*un String*), un **rôle** (*un String*), et un **email** (*un String*). Les employés sont des **Plain Old Java Object** (**POJO**), désignant des objets avec des attributs et des getters/setters.

On souhaite définir un service web REST permettant de manipuler cet ensemble de la manière suivante :

1. lister tous les employés ;
2. savoir le nombre des employés dans l'ensemble ;
3. récupérer un employé par son identifiant ;
4. ajouter un nouveau employé ;

5. supprimer un employé par son identifiant ;
6. modifier les informations d'un employé par son identifiant

De plus, une erreur survient si l'on essaye :

1. de récupérer/modifier/supprimer un employé dont l'identifiant n'existe pas comme identifiant valide d'un employé dans l'ensemble.
2. d'ajouter un employé avec un identifiant déjà affecté à un autre employé dans l'ensemble.

Les opérations sont détaillées davantage dans la table ci-dessous.

Définition	Paramètres	Description
`GET` employeeservice/ api / employees	pas de paramètres	lister tous les employés
`GET` employeeservice/ api / employees / count	pas de paramètres	savoir le nombre des employés dans l'ensemble
`GET` employeeservice/ api / employee / {id}	`id` (*obligatoire*) : l'identifiant de l'employé	récupérer un employé par son identifiant
`POST` employeeservice/ api / employees	pas de paramètres	ajouter un nouveau employé
`DELETE` employeeservice/ api / employee / {id}	`id` (*obligatoire*) : l'identifiant de l'employé	supprimer un employé par son identifiant
`PUT` employeeservice/ api / employee / {id}	`id` (*obligatoire*) : l'identifiant de l'employé	modifier les informations d'un employé par son identifiant

Une session de live coding sur cet exemple peut être consultée sur ce lien.

```
// ../source/rest/java/employee/api/Employee.java

ackage com.example.demo.models;

import java.util.Objects;

import javax.persistence.Entity;
```

```
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
@Entity
public class Employee {
    /* ATTRIBUTES */
    @Id
    @GeneratedValue
    private long id;
    private String name;
    private String role;
    private String email;

    public Employee() {

    }

    public Employee(String name, String role, String email) {
        this.name = name;
        this.role = role;
        this.email = email;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getRole() {
        return role;
    }

    public void setRole(String role) {
        this.role = role;
    }
```

```
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @Override
    public int hashCode() {
        return Objects.hash(email, id, name, role);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Employee other = (Employee) obj;
        return Objects.equals(email, other.email)
                && id == other.id && Objects.equals(name, other.name)
                && Objects.equals(role, other.role);
    }

    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" + name + ", role=" + role + ", email=" + ema
    }
}
// ../source/rest/java/employee/api/EmployeeRepository.java

ackage com.example.demo.repositories;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.models.Employee;

public interface EmployeeRepository extends JpaRepository<Employee, Long>{

}
// ../source/rest/java/employee/api/EmployeeData.java
```

```
ackage com.example.demo.data;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.example.demo.models.Employee;
import com.example.demo.repositories.EmployeeRepository;

@Configuration
public class EmployeeData {
    /* ATTRIBUTES */
    private Logger logger = LoggerFactory.getLogger(EmployeeData.class);

    @Bean
    public CommandLineRunner initDatabase(EmployeeRepository repository) {
        return args -> {
            logger.info("Preloading database with " + repository.save(
                    new Employee("John Doe", "CEO", "john.doe@example.com")));
            logger.info("Preloading database with " + repository.save(
                    new Employee("Jane Doe", "CTO", "jane.doe@example.com")));
        };
    }
}
```

```
// ../source/rest/java/employee/api/EmployeeController.java

ackage com.example.demo.controllers;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.exceptions.EmployeeNotFoundException;
import com.example.demo.models.Employee;
```

```java
import com.example.demo.repositories.EmployeeRepository;

@RestController
public class EmployeeController {
    /* ATTRIBUTES */
    @Autowired
    private EmployeeRepository repository;
    private static final String uri = "employeeservice/api";

    /* METHODS */
    @GetMapping(uri+"/employees")
    public List<Employee> getAllEmployees(){
        return repository.findAll();
    }

    @GetMapping(uri+"/employees/count")
    public String count() {
        return String.format("{\"%s\": %d}", "count", repository.count());
    }

    @GetMapping(uri+"/employees/{id}")
    public Employee getEmployeeById(@PathVariable long id) throws EmployeeNotFoundException
        return repository.findById(id)
                .orElseThrow(() -> new EmployeeNotFoundException(
                        "Error: could not find employee with ID " + id));
    }

    @ResponseStatus(HttpStatus.CREATED)
    @PostMapping(uri+"/employees")
    public Employee createEmployee(@RequestBody Employee employee) {
        return repository.save(employee);
    }

    @PutMapping(uri+"/employees/{id}")
    public Employee updateEmployee(@RequestBody Employee newEmployee,
            @PathVariable long id) {
        return repository.findById(id)
                .map(employee -> {
                    employee.setName(newEmployee.getName());
                    employee.setRole(newEmployee.getRole());
                    employee.setEmail(newEmployee.getEmail());
                    return repository.save(employee);
                })
                .orElseGet(() -> repository.save(newEmployee));
    }
```

```
    @ResponseStatus(HttpStatus.NO_CONTENT)
    @DeleteMapping(uri+"/employees/{id}")
    public void deleteEmployee(@PathVariable long id) throws EmployeeNotFoundException {
        Employee employee = repository.findById(id)
                .orElseThrow(() -> new EmployeeNotFoundException(
                        "Error: could not find employee with ID " + id));
        repository.delete(employee);
    }
}
```

```
// ../source/rest/java/employee/api/EmployeeException.java

ackage com.example.demo.exceptions;

public class EmployeeException extends Exception {
    public EmployeeException() {

    }

    public EmployeeException(String message) {
        super(message);
    }
}
```

```
// ../source/rest/java/employee/api/EmployeeNotFoundException.java

ackage com.example.demo.exceptions;

public class EmployeeNotFoundException extends EmployeeException {
    public EmployeeNotFoundException() {

    }

    public EmployeeNotFoundException(String message) {
        super(message);
    }
}
```

```
// ../source/rest/java/employee/api/EmployeeNotFoundExceptionAdvice.java

ackage com.example.demo.exceptions;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;
```

```
@ControllerAdvice
public class EmployeeNotFoundExceptionAdvice {

    @ExceptionHandler(EmployeeNotFoundException.class)
    @ResponseBody
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public String employeeNotFoundExceptionHandler(EmployeeNotFoundException e) {
        return String.format("{\"%s\": \"%s\"}", "error", e.getMessage());
    }
}
```

```
// ../source/rest/java/employee/api/EmployeeRestDemoApplication.java

ackage com.example.demo.main;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EntityScan(basePackages = {
        "com.example.demo.models"
})
@EnableJpaRepositories(basePackages = {
        "com.example.demo.repositories"
})
@SpringBootApplication(scanBasePackages = {
        "com.example.demo.data",
        "com.example.demo.exceptions",
        "com.example.demo.controllers"
})
public class EmployeeRestDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(EmployeeRestDemoApplication.class, args);
    }
}
```

```
// ../source/rest/java/employee/api/application.properties

spring.datasource.url=jdbc:h2:mem:employees
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
```

2.6 Consommer un service web REST avec Spring Boot

2.6.1 Workflow

1. Créer votre projet Spring Boot avec `Spring Starter Project` et choisissez les dépendances (*e.g., Spring Web*).
2. Créer les classes de votre modèle de données (*i.e., les ressources à consommer*).
3. Créer votre client RESTful qui encapsulera le proxy consommant le service Web.
4. Créer votre CLI.
5. Configurer votre application Spring Boot pour permettre au mécanisme de l'injection de dépendances de trouver vos entités, composants, configurations, ...
6. Configurer votre application dans `application.properties` afin de changer le numéro de port de votre serveur Tomcat, ...
7. Tester votre CLI.

2.6.2 Exemple d'une CLI consommant les opérations du service web REST exposant des informations sur des employés

Dans cet exemple, l'idée consiste à consommer le service web REST exposant des opérations sur un ensemble d'employés. Ces opérations seront utilisées par une CLI permettant à un client de les consommer. La CLI aura besoin ainsi de récupérer une instance du proxy du service web sur lequel elle invoquera l'opération choisie par le client.

La CLI est une application cliente héritant d'une classe abstraite `AbstractMain`. `AbstractMain` est une classe abstraite fournissant l'interface de base d'une CLI :

1. des attributs pour stocker l'URL du service web et terminer explicitement l'exécution de la CLI (*i.e., cette CLI continuera à s'exécuter tant que le choix la valeur de l'attribut `QUIT` n'a pas été choisie par le client*).
2. des méthodes pour saisir et vérifier l'URL du service web, mais aussi pour afficher le menu des interactions possibles.

Pour chaque CLI il y aura des méthodes supplémentaires pour : 1. récupérer une instance du proxy du service web ; 2. valider les valeurs saisies par l'utilisateur ; 3. invoquer les opérations correspondantes du service web avec éventuellement les valeurs saisies validées comme paramètres.

Le proxy est une instance de la classe `RestTemplate` construite par une instance de la `RestTemplateBuilder` en utilisant le mécanisme d'injection de dépendances automatique fourni par Spring Boot.

De plus, la validation des valeurs saisies par l'utilisateur est réalisée par

un processeur d'input défini dans la hiérarchie de classes enracinée par `ComplexUserInputProcessor<T>`. Par exemple, pour traiter une valeur saisie qui devrait correspondre à un entier ou à un entier non nulle, la CLI utilise une instance de `IntegerInputProcessor` ou `NonZeroIntegerInputProcessor` respectivement.

N.B. En général, une CLI n'a pas forcément besoin de suivre l'interface définie par `AbstractMain` ou d'utiliser un processeur d'input utilisateur. Il s'agit de concepts créés pour faciliter la réutilisation de code et simplifier la redondance, mais vous n'êtes pas forcément obligés de s'en servir ou d'en définir vous-mêmes.

Une session de live coding sur cet exemple peut être consultée sur ce lien.

```
// ../source/rest/java/employee/client/Employee.java

ackage com.example.demo.models;

import java.util.Objects;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@JsonIgnoreProperties(ignoreUnknown = true)
public class Employee {
    /* ATTRIBUTES */
    private long id;
    private String name;
    private String role;

    /* CONSTRUCTORS */
    public Employee() {

    }

    public Employee(String name, String role) {
        this.name = name;
        this.role = role;
    }

    /* METHODS */
    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getRole() {
        return role;
    }

    public void setRole(String role) {
        this.role = role;
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, name, role);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Employee other = (Employee) obj;
        return id == other.id && Objects.equals(name, other.name) && Objects.equals(role, ot
    }

    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" + name + ", role=" + role + "]";
    }
}
// ../source/rest/java/employee/client/EmployeeServiceClient.java

ackage com.example.demo.client;

import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;
```

```
import org.springframework.web.client.RestTemplate;

@Component
public class EmployeeServiceClient {
    /* METHODS */
    @Bean
    public RestTemplate generateRestTemplate(RestTemplateBuilder builder) {
        return builder.build();
    }
}
```

```
// ../source/rest/java/employee/client/AbstractMain.java

ackage com.example.demo.cli;

import java.io.BufferedReader;
import java.io.IOException;

public abstract class AbstractMain {
    public static String SERVICE_URL;
    public static final String QUIT = "0";

    protected void setTestServiceUrl(BufferedReader inputReader)
            throws IOException {

        System.out.println("Please provide the URL to the web service to consume: ");
        SERVICE_URL = inputReader.readLine();

        while(!validServiceUrl()) {
            System.err.println("Error: "+SERVICE_URL+
                    " isn't a valid web service WSDL URL. "
                    + "Please try again: ");
            SERVICE_URL = inputReader.readLine();
        }
    }

    protected abstract boolean validServiceUrl();

    protected abstract void menu();
}
```

```
// ../source/rest/java/employee/client/ComplexUserInputProcessor.java

ackage com.example.demo.cli;

import java.io.BufferedReader;
import java.io.IOException;
```

```java
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.function.Predicate;

public abstract class ComplexUserInputProcessor<T> {
    /* ATTRIBUTES */
    protected String message;
    protected BufferedReader inputReader;
    protected Method parser;
    protected Predicate<String> isValid;
    protected T parameter;

    /* CONSTRUCTOR */
    public ComplexUserInputProcessor(BufferedReader inputReader) {
        this.inputReader = inputReader;
        setMessage();
        setParser();
        setValidityCriterion();
    }

    /* METHODS */
    protected abstract void setMessage();
    protected abstract void setValidityCriterion();
    protected abstract void setParser();

    public T process() throws IOException {
        System.out.println(message);
        String userInput = inputReader.readLine();

        while (!isValid.test(userInput)) {
            System.err.println("Sorry, wrong input. Please try again.");
            System.out.println();
            System.out.println(message);
            userInput = inputReader.readLine();
        }

        try {
            parameter = (T) parser.invoke(null, userInput);
        } catch (SecurityException | IllegalAccessException |
                IllegalArgumentException | InvocationTargetException e) {

            e.printStackTrace();
        }

        return parameter;
    }
```

```
}
// ../source/rest/java/employee/client/IntegerInputProcessor.java

ackage com.example.demo.cli;

import java.io.BufferedReader;

public class IntegerInputProcessor extends ComplexUserInputProcessor<Integer> {

    public IntegerInputProcessor(BufferedReader inputReader) {
        super(inputReader);
    }

    @Override
    protected void setMessage() {
        message = "Please enter an integer:";
    }

    @Override
    protected void setValidityCriterion() {
        isValid = str -> {
            try {
                Integer value = Integer.parseInt(str);
                return value instanceof Integer;
            } catch (NumberFormatException e) {
                return false;
            }
        };
    }

    @Override
    protected void setParser() {
        try {
            parser = Integer.class.getMethod("parseInt", String.class);
        } catch (SecurityException | NoSuchMethodException e) {

            e.printStackTrace();
        }
    }
}
// ../source/rest/java/employee/client/EmployeeRestClientCLI.java

ackage com.example.demo.cli;

import java.io.BufferedReader;
```

```java
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.client.RestTemplate;

import com.example.demo.models.Employee;
import com.fasterxml.jackson.databind.ObjectMapper;

@Component
public class EmployeeRestClientCLI extends AbstractMain implements CommandLineRunner {
    /* ATTRIBUTES */
    @Autowired
    private RestTemplate proxy;
    private IntegerInputProcessor inputProcessor;
    private static String URI_EMPLOYEES; // <service_url>/employees
    private static String URI_EMPLOYEES_ID; // <service_url>/employees/{id}

    /* METHODS */
    @Override
    public void run(String... args) throws Exception {
        BufferedReader inputReader;
        String userInput = "";

        try {
            inputReader = new BufferedReader(
                    new InputStreamReader(System.in));
            setTestServiceUrl(inputReader);
            URI_EMPLOYEES = SERVICE_URL + "/employees";
            URI_EMPLOYEES_ID = URI_EMPLOYEES + "/{id}";

            do {
                menu();
                userInput = inputReader.readLine();
                processUserInput(inputReader, userInput, proxy);
                Thread.sleep(3000);

            } while(!userInput.equals(QUIT));
        } catch (IOException e) {
            e.printStackTrace();
```

```
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

}

@Override
protected boolean validServiceUrl() {
    return SERVICE_URL.equals(
            "http://localhost:8080/employeeservice/api");
}

@Override
protected void menu() {
    StringBuilder builder = new StringBuilder();
    builder.append(QUIT+". Quit.");
    builder.append("\n1. Get number of employees.");
    builder.append("\n2. Display all employees.");
    builder.append("\n3. Get employee by ID");
    builder.append("\n4. Add new employee");
    builder.append("\n5. Remove employee by ID");
    builder.append("\n6. Update existing employee");

    System.out.println(builder);
}

private void processUserInput(BufferedReader reader,
        String userInput, RestTemplate proxy) {
    Map<String, String> params = new HashMap<>();
    inputProcessor = new IntegerInputProcessor(reader);
    try {
        switch(userInput) {
            case "1":
                String uri = URI_EMPLOYEES + "/count";
                String countStr = proxy.getForObject(uri, String.class);
                ObjectMapper mapper = new ObjectMapper();
                long count = (int) mapper.readValue(countStr, Map.class).get("count");
                System.out.println(String.format("There are %d employees", count));
                System.out.println();
                break;

            case "2":
                uri = URI_EMPLOYEES;
                Employee[] employees = proxy.getForObject(uri, Employee[].class);
                System.out.println("Employees:");
                Arrays.asList(employees)
```

```
                .forEach(System.out::println);
                System.out.println();
                break;

            case "3":
                uri = URI_EMPLOYEES_ID;
                System.out.println("Employee ID:");
                int id = inputProcessor.process();
                params.put("id", String.valueOf(id));
                Employee employee = proxy.getForObject(uri, Employee.class, params);
                System.out.println(String.format("Employee with ID %s: %s", id, employee
                System.out.println();
                break;

            case "4":
                uri = URI_EMPLOYEES;
                System.out.println("Employee Name:");
                String name = reader.readLine();
                System.out.println();

                System.out.println("Employee Role:");
                String role = reader.readLine();
                System.out.println();

                Employee createdEmployee = new Employee(name, role);
                Employee returnedEmployee = proxy.postForObject(uri, createdEmployee, Em
                System.out.println(String.format("Successfully added employee: %s", retu
                System.out.println();
                break;

            case "5":
                uri = URI_EMPLOYEES_ID;
                System.out.println("Employee ID:");
                id = inputProcessor.process();
                params.put("id", String.valueOf(id));
                proxy.delete(uri, params);
                System.out.println(String.format("Successfully removed employee with ID
                System.out.println();
                break;

            case "6":
                uri = URI_EMPLOYEES_ID;
                System.out.println("Employee ID:");
                id = inputProcessor.process();
                System.out.println("New Employee Name:");
                name = reader.readLine();
```

```
                    System.out.println();

                    System.out.println("New Employee Role:");
                    role = reader.readLine();
                    System.out.println();

                    params.put("id", String.valueOf(id));
                    Employee newEmployee = new Employee(name, role);

                    proxy.put(uri, newEmployee, params);
                    System.out.println(String.format("Successfully updated/created employee
                    System.out.println();
                    break;

                case QUIT:
                    System.out.println("Bye...");
                    System.exit(0);

                default:
                    System.err.println("Sorry, wrong input. Please try again.");
                    return;
            }
        } catch (IOException e) {
            e.printStackTrace();
        } catch (HttpClientErrorException e) {
            System.err.println(e.getMessage());
            System.err.println("Please try again with a different ID.");
        }
    }
}
```

```
// ../source/rest/java/employee/client/EmployeeRestClientDemoApplication.java

ackage com.example.demo.main;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication(scanBasePackages = {
        "com.example.demo.models",
        "com.example.demo.client",
        "com.example.demo.cli"
})
public class EmployeeRestClientDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(EmployeeRestClientDemoApplication.class, args);
```

```
    }
}
```

```
// ../source/rest/java/employee/client/application.properties

server.port=8084
```

2.7 Références et liens utiles

1. A Solid Guide to SOLID Principles
2. Frameworks et réutilisation
3. Schémas et patrons de conception
4. Difference between “Inversion of Control”, “Dependency inversion” and “Decoupling”
5. IoC vs. DIP vs. DI vs. IoC Container
6. Building REST services with Spring
7. Consuming a RESTful Web Service
8. Spring Boot RestTemplate for RESTful clients example
9. How To Write REST Consumer API Using Spring Boot
10. REST PUT resource with ID while having auto-generated IDs