Spring Boot & REST

Plan

- Protocole REST
- Réagir à des requêtes HTTP
- Communiquer avec d'autres WebServices
- Retour sur la programmation défensive
- Appartée sur la journalisation

Spring Boot & REST

Protocole REST

REST

REST (representational state transfer) : “un style d'architecture logicielle définissant un ensemble de contraintes à utiliser pour créer des services web.” Wikipedia

Par abus de langage, on parle souvent de REST en tant que protocole de communication dans une relation client/serveur.

REST

Une histoire de ressources

REST place la notion de “**Ressource”** au centre du modèle de développement.

Ressource/Resource: Représentation textuelle, ou plus généralement binaire, d’un objet ou d’un concept, qui peut être identifiée.

Exemple de ressource : Un élève, une réservation de marché aux puces,

Une citation, etc

REST

Un Web Service REST

On parle de WebService REST pour une application permettant la manipulation de ces Ressources, au travers d'un ensemble fini et connu d'opérations **sans état/stateless** via des requêtes HTTP.

Dans ce contexte, les ressources sont identifiées par une URL qui leur est propre.

Exemple: http://monappli.com/articles/1

Rappel

Stateful...

Dans le cas d'un traitement Stateful, l'application se souvient du client qui effectue la requête.

Il va chercher dans le scope Session ou dans le scope WebContext des éléments précédemment enregistrés.

Exemple: un historique de navigation ou d'appels

Rappel

… Ou Stateless

Dans le cas d'un traitement Stateless, la requête contient tous les éléments nécessaires à son traitement.

Le seule scope utilisé est celui de la requête.

Exemple: une recherche

REST

Un Web Service REST

Les opérations de manipulation des ressources sont définies par au moins ces éléments :

- Une **URL** : http://monappli.com/articles/1. En règle général, le pluriel est utilisé dans les URLs.
- Une **méthode HTTP** : GET, POST, DELETE, etc

La méthode va donner la nature de l'opération

Un Web Service REST

D’autres éléments peuvent être ajoutées à la requête HTTP pour préciser le comportement que doit suivre l’opération

- Un **body/corps** de requête : il peut contenir toute sorte de données nécessaires
- Un ou plusieurs **headers HTTP** : des éléments d’authentification, des informations sur la langue souhaitée ou la forme du contenu
- Des **paramètres** de requête: ils donnent des précisions sur l’attendu du client

REST

Contenu des échanges

En règle général, le contenu des échanges entre un client et un serveur dans un protocole REST sera formalisée en **Json**, issu du langage Javascript.

Toutefois, rien n'oblige ce formalise. Il est tout à fait possible d'utiliser du XML, du texte plat, etc.

Json

Exemple d'objet Json

```
{   "name": "test",
    "property": 1,
    "boolean": true,
    "subProperty": {
        "array": [ "1", "2" ]
    }
}
```

L'indentation n'a pas d'importance.

Seule la structure compte.

REST

Précision sur le format

Dans un Web Service idéal, le contenu sera adapté à la requête faite par le client.

Le Header Http **“Accept”** permet au client de préciser le format attendu de la réponse.

Le Header Http **“Content-type”** de la réponse serveur *devrait* utiliser le format attendu par la requête.

REST et CRUD

C.R.U.D.

Il s'agit des actions de base que l'on peut effectuer sur une entité

- **C**reate : Création d'une nouvelle entité
- **R**ead : Lecture des entités
- **U**pdate : Mise à jour des entités
- **D**elete: Suppression des entités

Dans le protocole REST, toute opération est **AU PLUS** une combinaison de plusieurs de ces actions

REST et CRUD

Méthodes HTTP associées

Création	**POST**
Mise à jour complète	**PUT**
Mise à jour partielle	**PATCH**
Lecture	**GET**
Suppression	**DELETE**

REST et CRUD

Exceptions

Certains actions sont parfois difficiles à associer à une opération et une ressource.

Exemple : Envoi d'un mail, action métier impliquant plusieurs ressources, etc

Deux approches sont possibles face à cette situation

REST et CRUD

1. **Trouver une nouvelle ressource**

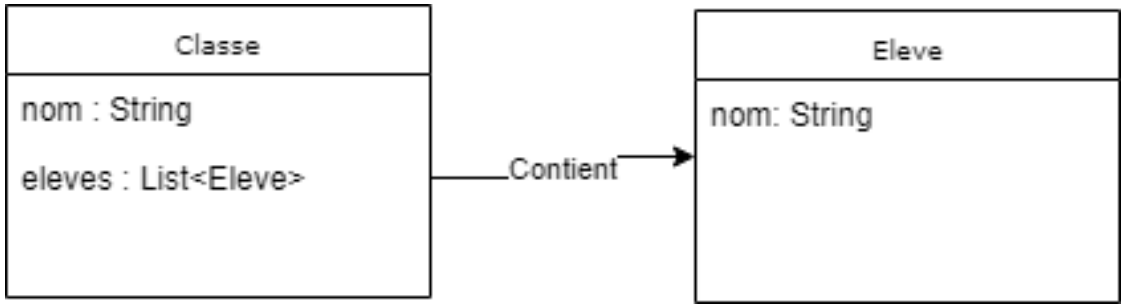
   Les mails ne peuvent-ils pas être une ressource REST comme une autre?

1. **Tordre le protocole**

   C’est la solution de facilité déconseillée mais souvent rencontrée: une ressource POST fourre tout

Cas pratique

L'exemple qui suit est basé sur ce modèle :



Une classe contient n élèves

Le nom de la classe est son identifiant

Le nom de l'élève est son identifiant

Cas pratique : CRUD

Créer une classe

URL: /classes

Méthode HTTP : POST

Corps: les données de la classe en JSON

Récupérer une classe

URL: /classes/<nom de la classe> (/classes/CE1)

Méthode HTTP : GET

Corps: rien

Cas pratique : CRUD

Modifier une classe

URL: /classes/<nom de la classe>

Méthode HTTP : PUT

Corps: les données complètes de la classe en JSON

Modifier partiellement une classe

URL: /classes/<nom de la classe>

Méthode HTTP : PATCH

Corps: les données modifiées de la classe en JSON

Cas pratique : CRUD

- **Supprimer une classe**

  URL: /classes/<nom de la classe>

  Méthode HTTP : DELETE

  Corps: rien

- **Récupérer la liste de toutes les classes**

  URL: /classes

  Méthode HTTP : GET

  Corps: rien

Cas pratique : CRUD

Récupérer les élèves d'une classe

URL: /classes/<nom de la classe>/eleves

Méthode HTTP : GET

Corps: rien

Les élèves sont ici une sous ressource de la classe. Cela illustre le lien entre ces deux entités.

Cas pratique : CRUD

Lier une classe à des élèves

URL: /classes/<nom de la classe>

Méthode HTTP : PUT ou PATCH

Corps: La classe complète ou partielle en JSON, avec au moins la liste des élèves

Remarques sur les URLs:

- Le dernier fragment d'une URL doit toujours être une ressource ou un identifiant
- L'entité retournée retournée doit toujours être de même type que la dernière ressource présente dans l'URL

"/ressource/1/sousressource/2" doit retourner une "sousressource" et pas une "ressource"

- Il est possible d'accéder à la même ressource via plusieurs URLs.

Importance du statut HTTP

Une information complémentaire

En plus des headers et du corps, une réponse HTTP comporte un statut.

Il s'agit d'un nombre normalement compris entre 100 et 527.

Il donne une indication sur la validité de la réponse, de la requête ou même d'une erreur.

Importance du statut HTTP

1xx	**Informations**
2xx	**Succès**
3xx	**Redirection**
4xx	**Erreur du client**
5xx	**Erreur du serveur**

Importance du statut HTTP

Status les plus courants

200	La requête a été traitée avec succès
404	La ressource demandée n'existe pas
401	Le client n'est pas autorisé à accéder à cette ressource
204	La réponse ne contient rien (body vide)
500	Le serveur a rencontré une erreur qui l'a empêché de répondre

Résumé

Condensé des bonnes pratiques

https://www.restapitutorial.com/lessons/httpmethods.html

Spring Boot & REST

Réagir à des requêtes HTTP

Servlets

Des bases dans JEE

JEE fournit l'API des **Servlets** : des classes qui interagissent avec des requêtes HTTP.

Spring définit une ou plusieurs Servlets pour être capable de "servir" les requêtes qui parviennent à l'application dans laquelle il est intégrée.

Contrairement à JEE classique ou même à d'autres libraires (CXF, axis), Spring Boot va masquer cette mécanique : elle ne présente que peu d'intérêts à configurer.

Auto Configuration

Spring-boot-starter-web

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>2.0.5</version>
    </dependency>
</dependencies>
```

Auto Configuration

Spring-boot-starter-web

Via la mécanique d'Auto Configuration, la présence des classes de la dépendance **spring-boot-starter-web** dans le classpath de l'application va entraîner la création de plusieurs beans dans le contexte Spring.

Parmis ces éléments se trouvera une instance DispatcherServlet, servlet Spring qui sert à intercepter les requêtes parvenant à l'application.

Auto Configuration

Comment l'activer?

Par défaut, l'auto configuration est inactive. Elle peut être activée par le biais d'annotations à ajouter sur la Main Class de l'application Spring Boot.

```
@SpringBootApplication
```

OU

```
@EnableAutoConfiguration
```

Il s'agit d'annotations "class-level" : elles se posent sur la définition de classe

Configuration Manuelle

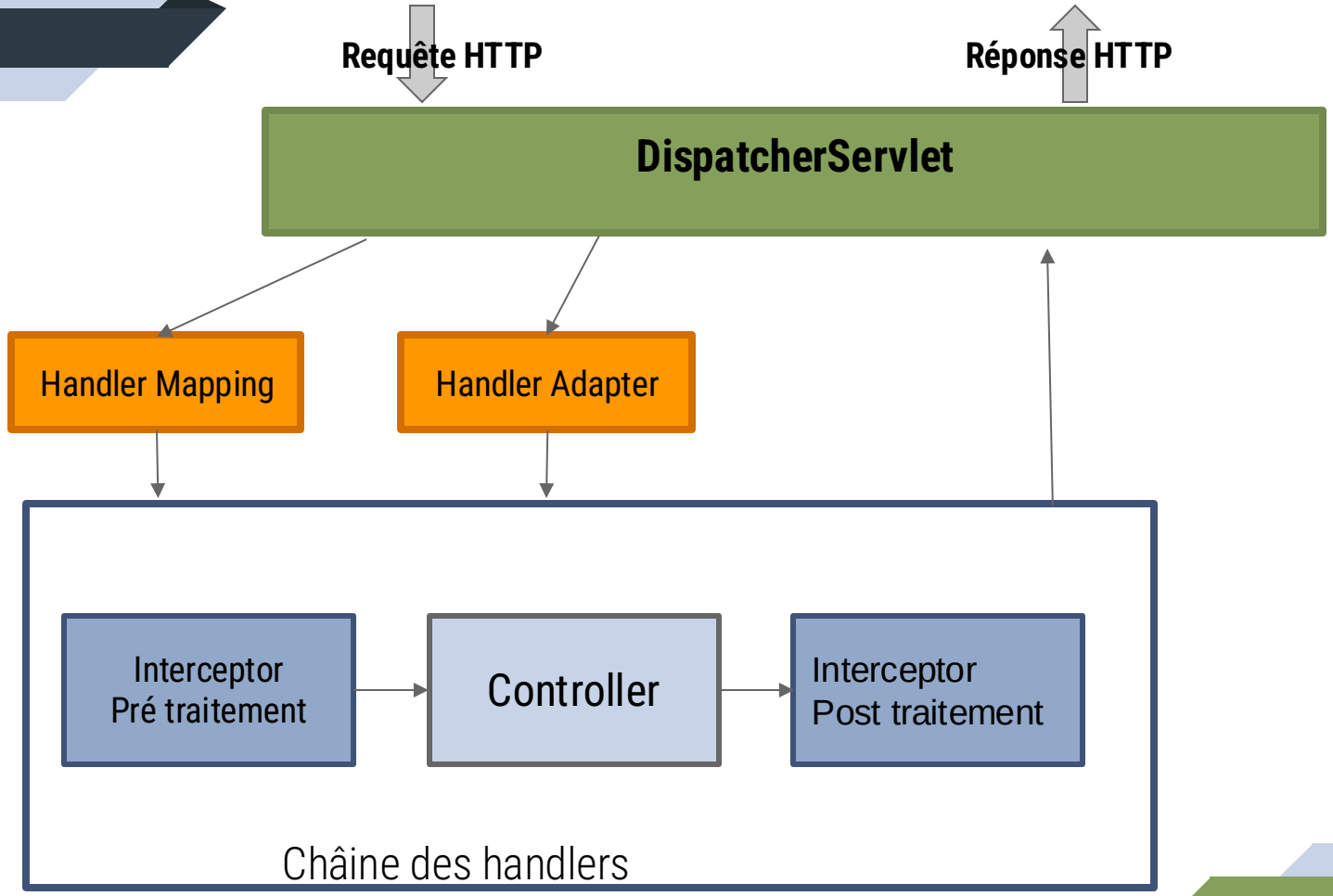
Configuration Java

L'auto configuration peut être désactivée, ou absente. La configuration en Java de la servlet se ferait comme suit :

```
@Bean
public ServletRegistrationBean exampleServletBean() {
    ServletRegistrationBean bean = new ServletRegistrationBean(
            new DispatcherServlet(), "**");
    bean.setLoadOnStartup(1);
    return bean;
}
```

Configuration Manuelle (web.xml, hors Spring Boot)

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/dispatcher-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```


Requête HTTP
Réponse HTTP
DispatcherServlet
Handler Mapping
Handler Adapter
Interceptor
Pré traitement
Controller
Interceptor
Post traitement
Châine des handlers

Controller

Définition

Les **Controllers** sont les classes en charge de réagir à une requête spécifique, en fonction de son URL, de sa méthode HTTP et éventuellement d'autres paramètres.

Il s'agit de classes Java identifiées par des annotations spécifiques.

Ils sont en charge du lien entre la ressource REST, son URL, et le traitement des différentes opérations associées.

Controller

Exemple

```
@Controller
@ResponseBody
public class TestController {

    @RequestMapping(method = RequestMethod.GET, path = "/myresources")
    public List<MyResource> getAllResources(){...}
}
```

Controller

Annotations

`@Controller :` Cette annotation class-level identifie la classe comme un Controller. Une instance de cette classe sera ajoutée au contexte Spring. Par défaut, ce sont des singletons.

On parle de **stéréotype Spring**: une classe dont le rôle est identifié grâce à son annotation.

Cette annotation est une extension de l'annotation `@Component.`

Controller

Réagir à une requête

`@RequestMappging` permet à une méthode d'un `@Controller` d'être exécutée lorsqu'une requête HTTP respectant les critères spécifiés est reçue par l'application.

```
@RequestMapping(method = RequestMethod.GET, path = "/test")
```

Permettra d'exécuter une méthode lorsqu'un appel GET sera reçu sur l'URL <MonApplication>/test

Controller

Réagir à une requête

Les paramètres de l'annotation `@RequestMapping` précisent les conditions de réaction.

`Method` : Méthode HTTP sur laquelle réagir

`Headers` : Header HTTP sur lequel réagir

`Path` : Fragment d'URL sur lequel réagir

`Params` : Paramètres de la requête sur lesquels réagir

Si le nom du paramètre n'est pas spécifié, il s'agit de Path.

Controller

Réagir à une requête

`@RequestMapping` peut être utilisé en tant qu'annotation class-level, sur un controller.

Utilisée à cet endroit, elle fait généralement le lien entre une ressource REST et tout un controller.

On n'y précise souvent que la racine de l 'URL.

Controller

Réagir à une requête

`@RequestMapping` possède des Alias pour les méthodes HTTP les plus courantes.

`@RequestMapping(method = RequestMethod.GET)` est ainsi équivalent à `@GetMapping`

Il existe ainsi `@GetMapping, @PostMapping, @DeleteMapping` ,...

Controller

Exemple V2

```
@Controller
@RequestMapping("/myresources")
public class TestController {

    @GetMapping(path = "")
    public List<MyResource> getAllResources(){... }
}
```

Controller

DeSerialization

Les controller manipulent dans leur signature et dans le corps de leur méthode des objets Java, souvent reflets des éléments contenus dans le corps de la requête ou de la réponse.

Ce contenu prend en règle général la forme d'une trame Json et donc d'un texte.

Le processus de conversion du Json vers un Objet Java est appelé **Désérialisation**.

Serialization

Le procédé inverse, à savoir transformer un objet Java en sa représentation Json est appelé **Sérialisation**.

On regroupe parfois ces deux procédés sour le nom de **Marshalling**

Ce fastidieux travail est effectué par défaut dans Spring Boot par la librairie Jackson.

Json < - > Java

Json

```
{
 "name": "test",
 "id": 1,
 "properties":["small", "tall"]
}
```

Objet Java

```
public class Test {
    private String name;
    private Integer id;
    private List<String> properties;
}
```

Mécanisme par défaut

Un comportement par défaut très simple...

Tel que configuré avec l'auto configuration, Jackson cherche à faire un mapping comme suit :

Un objet Java doit avoir une propriété de même nom que la clef Json.

Un objet Json doit correspondre à un Objet Java

Les types simples sont gérés nativement: chaînes, entiers, etc.

Mécanisme par défaut

`@JsonIgnore` permet d'ignorer une propriété par lors de la sérialisation / déserialisation

`@JsonIgnoreType` permet d'ignorer toute une classe par lors de la sérialisation / déserialisation. C'est une annotation class-level.

`@JsonProperty("<clef>")` permet de mapper spécifiquement une propriété de l'objet sur la clef Json "<clef>"

Mécanisme par défaut

...Pénible à surcharger

Le changement du mapping par défaut se fait via l'écriture de classe type "Serializer" et "Deserializer".

Ces classes doivent alors manipuler le Json directement, au moyen d'objets et d'une API fournis par Jackson.

Controller

Annotations (Bis)

`@ResponseBody :` Cette annotation class-level ou method-lvl spécifie que le retour de la méthode doit être ajoutée au corps d'une HttpResponse, après une étape de sérialisation assurée par Jackson.

Cette annotation ainsi que `@Controller` peuvent être remplacés par une autre annotation regroupant les 2 : `@RestController`.

@RestController

définit les points d’entrée de chaque ressource

Paramètres d'une requête

Depuis l'URL

Si l'URL d'une requête est "*/maRessource /1*" , "1" est l'identifiant de la ressource que l'on souhaite manipuler

Il peut être intéressant de le récupérer au sein du traitement depuis le controller pour pouvoir le cibler spécifiquement

Paramètres d'une requête

```
@GetMapping("/maRessource/{id}")
public MaRessource getResource(@PathVariable("id") Integer id){ }
```

Cette récupération se fait en deux temps:

1. Variabiliser l'URL définie dans le `@RequestMapping` ou son alias

1. Récupérer cette variable dans les paramètres de la méthode via l'annotation `@PathVariable.` Le paramètre de l'annotation est le nom de l'attribut

Paramètres d'une requête

Il est possible de récupérer plusieurs paramètres depuis l'URL

```
@GetMapping("/maRessource/{id}/maSousRessource/{id2}")

public MaSousRessource getSousResource(@PathVariable("id") Integer
id, @PathVariable("id2") Integer id2){ }
```

Paramètres d'une requête

Depuis les paramètres de l'URL

Si l'URL d'une requête est *"/maRessource?val=test"* , "test" est la valeur du paramètre "val", qui doit avoir une influence sur le traitement

```
@GetMapping("/maRessource")
public MaRessource getAllResources(@RequestParam("val") String val){
}
```

L'annotation `@RequestParam` permet de récupérer la valeur des paramètres de l'URL.

Paramètres d’une requête

Si de nombreux paramètres sont à récupérer depuis l’URL, un objet peut être passé en paramètre de la méthode, sans annotation.

Si ces propriétés ont les mêmes types et noms que les paramètres, l’objet contiendra les valeurs des paramètres.

Paramètres d'une requête

```
@GetMapping("/maRessource”)
public MaRessource getAllResources(@RequestParam("val”) String val,
@RequestParam("val2”) String val2){ }
```

Équivaut à

```
@GetMapping("/maRessource”)
public MaRessource getAllResources(Parameters parameters){ }
```

Paramètres d'une requête

Si et seulement si Parameters est défini comme suit:

```
public class Parameters {
    private String val2;
    private String val;

    //Accesseurs
}
```

Paramètres d'une requête

Depuis les headers

```
@GetMapping("/maRessource")

public MaRessource getAllResources(@HeaderParam("header") String
header){ }
```

L'annotation `@HeaderParam` permet de récupérer la valeur des headers de la requête HTTP

Paramètres d'une requête

Depuis le corps de la requête

```
@PostMapping("/maRessource”)

public MaRessource getAllResources(@BodyParam MaRessource
maRessource){ }
```

Le corps de la requête étant généralement écrit au format Json, Jackson va se charger de désérialiser son contenu pour le convertir en Objet.

La récupération se fait via l'annotation `@BodyParam`

Paramètres d'une requête

Combinaison

```
@PostMapping("/maRessource”)

public MaRessource getAllResources(@BodyParam MaRessource
maRessource, @HeaderParam("header”) String header,
@RequestParam("val”) String val, @PathVariable("id") Integer id){ }
```

Il est possible de combiner l'ensemble de ces annotations dans les paramètres d'une méthode

Gestion des status HTTP

Cas nominal

Par défaut, Spring Boot va correctement ajouter le bon statut HTTP à la réponse HTTP:

- 200 si tout s'est bien passé
- 204 si le corps de la requête est vide
- 500 si une exception non catchée est rencontrée
- ...

Gestion des status HTTP

Via Response Entity

Un statut spécifique peut être retourné par un controller, à condition que la signature de la méthode retourne un objet de type **ResponseEntity**.

Il s'agit d'un wrapper du contenu de la réponse, auquel on peut passer un contenu, un statut d'erreur, etc.

```
@RequestMapping("/maRessource")
public ResponseEntity sendViaResponseEntity() {
    return new ResponseEntity(HttpStatus.NOT_ACCEPTABLE);
}
```

Gestion des status HTTP

Via Response Entity

Un body peut être ajouté dans la réponse.

On utilisera alors le pattern builder, et la classe **BodyBuilder**, retournée par la méthode `.status()`.

La méthode `.body(objet_du_body)` retourne une instance de ResponseEntity<?> où ? est le type de l'objet en paramètre.

```
return
ResponseEntity.status(HttpStatus.BAD_GATEWAY).body(monBody);
```

Gestion des status HTTP

Via Exceptions

Une exception spécifique peut être levée dans le code. Si annotée correctement, Spring peut convertir l'exception en un code HTTP associé.

```
@RequestMapping("maRessource")
public MaRessource sendViaException() {
    throw new CustomException();
}
```

Gestion des statuts HTTP

L'annotation `@ResponseStatus` class-level doit être ajoutée à la définition de la classe de l'Exception.

```
@ResponseStatus(HttpStatus.FORBIDDEN)
public class CustomException extends Exception {}
```

Gestion des statuts HTTP

Un message spécifique peut être renvoyé dans le corps de la réponse. Il est à préciser dans l'annotation avec l'attribut reason.

```
@ResponseStatus(HttpStatus.FORBIDDEN, reason = "Message d'erreur")
public class CustomException extends Exception {}
```

Pour les versions récentes de Spring Boot (>2.3.1), une ligne doit être ajoutée au fichier application.properties:

```
server.error.include-message=always
```

Spring Boot & REST

Communiquer avec d’autres WebServices REST

Pourquoi?

Une application de Web Service Rest peut être amenée à consommer d'autres Web Services REST (ou autres).

Elle peut avoir besoin d'informations complémentaires, détenues dans d'autres SI, s'imbriquer dans une architecture MicroServices, utiliser des informations publiques, etc.

Un client REST

Instanciation

La classe **org.springframework.web.client.SpringRestTemplate** permet d' interroger un Web Service REST

Une instance peut être créée via

- Une instanciation classique : `new RestTemplate();`
- La classe `RestTemplateBuilder.build()`. Cette seconde approche permet un paramétrage commun entre plusieurs RestTemplate (intercepteur, sérialisation)

Un client REST

Instanciation

Le `RestTemplateBuilder` est un objet du contexte Spring.

Il ne **DOIT PAS** être instancié à la main

```
new RestTemplateBuilder()
```

Il **DOIT** toujours être injecté

```
@Autowired
RestTemplateBuilder restTemplateBuilder;
```

Appel GET

```
ResponseEntity<MaRessource> response =
        new RestTemplate().getForEntity("url/resource/rest",MaRessource.class);
```

OU

```
MaRessource maRessource =
         new RestTemplate().getForObject("url/resource/rest",MaRessource.class);
```

Appel GET

Paramètres de l'appel

Ces deux méthodes prennent en premier paramètre l'URL d'appel et en second paramètre le type de l'objet renvoyé par la requête.

Ce peut être un Type simple (String,...) ou un Pojo.

Dans ce second cas, Jackson va intervenir pour assurer le procédé de sérialisation et désérialisation.

L'approche REST encourage très fortement à utiliser des Pojo, chacun d'entre eux étant plus ou moins la matérialisation d'une ressource.

Object vs Entity

`getForEntity` retourne un objet pramétrable de type `ResponseEntity`.

Des informations complémentaires sont accessibles via cet objet (statut HTTP), mais un travail de récupération supplémentaire du contenu est à prévoir.

`getForObject` retourne directement une instance de la classe passée en second paramètre.

Les statuts d'erreurs devront être gérés via une mécanique d'exceptions

Object vs Entity

Cette différenciation entity/object va exister pour toutes les méthodes HTTP classiques : GET, POST, DELETE, …

Appel GET

Usage des ResponseEntity

`.getBody()` retourne par défaut un Object.

Lorsque la ResponseEntity est typée, la méthode retourne un objet du type paramétré

```
new ResponseEntity<String>(HttpStatus.BAD_GATEWAY).getBody()
```

retourne un objet de type String.

Appel GET

Usage des ResponseEntity

- `getStatusCode()` retourne le status code de la réponse sous la forme de l'énumération HttpStatus. Elle possède un attribut value contenant la valeur numérique du statut.
- `getStatusCodeValue()` retourne directement la valeur numérique du statut.

- `getHeaders()` retourne un objet de type `HttpHeaders`. Sur ce dernier peuvent être récupérés les headers via la méthode `get()`. Il existe des méthodes d'accès pour les headers les plus courants. Exemple : `getAcceptLanguage()`

Autres méthodes

POST

```
new RestTemplate().postForObject("url/resource/rest",
    new MaRessource(),MaRessource.class);
```

Dans le cas des méthodes pouvant contenir un corps de requête, les paramètres sont, dans l'ordre :

1. URL d'appel
2. Corps de la requête, à représenter sous la forme d'un Pojo
3. Type de retour de l'appel

Autres méthodes

PATCH

```
new RestTemplate().patchForObject("url/resource/rest",
      new MaRessource(),MaRessource.class);
```

La méthode est rigoureusement la même que pour le POST.

(NB: Dans un respect strict du protocole REST, le type en corps de requête devrait être le même que celui en retour)

Autres méthodes

DELETE

```
new RestTemplate().delete("url/resource/rest");
```

En principe, un appel DELETE devrait ne rien retourner, hormis un status HTTP précisant la réussite ou non de l'appel.

Autres méthodes

PUT

```
new RestTemplate().put("url/resource/rest", new MaRessource());
```

Cette méthode ne renvoie rien!

Autres méthodes

PUT avec un retour

```
ResponseEntity<MaRessource> maRessource =
        new RestTemplate().exchange("url/resource/rest", HttpMethod.PUT, new
MaRessource(), MaRessource.class);
```

La méthode exchange est la mécanique derrière les méthodes getForEntity, postForObject,...

Autres méthodes

Exchange

Il est possible de remplacer toutes les méthodes précédentes par la méthode `.exchange(...)`

Ordre des paramètres :

1. L' URL d'appel
2. La méthode HTTP
3. Le corps de la requête, qui peut donc être null
4. Le type de retour

Appartée

Properties

Toutes les valeurs contenues dans le fichier application.properties peuvent être lues et utilisées dans l'application.

```
@Value("${ma.propriete}")

private String inceptionUrl;
```

Timeout

Attente utilisateur

Le but des WebServices REST est de servir une donnée à un client, le tout dans un temps *raisonnable*.

Chaque client peut avoir un délai de tolérance variable. Si ce délai est dépassé, on dit que la requête part en “timeout” :

- Le client n’attend plus le résultat.
- Si le résultat parvient un jour au client, il ne l’utilisera sans doute pas

Timeout

Appliquer un Timeout

Il est possible d'appliquer un délai de Timeout à un RestTemplate, via le RestTemplateBuilder.

```
restTemplateBuilder
    .setConnectTimeout(1000)
    .setReadTimeout(1000)
    .build();
```

Les délais sont en millisecondes.

Timeout

Gérer le timeout

Lorsqu'un appel dépasse les délais configurés, une exception sera levée.

Elle pourra

- Etre propagée, auquel cas le client final de l'application recevra une erreur, probablement avec un code HTTP 500
- Etre récupérée puis intégrée à mécanisme de redressement afin de fournir une réponse, même dégradée, au client: un **fallback**

Spring Boot & REST

Retour sur la programmatoin défensive

Rappel

Contrôler les entrées !

L'une des préconisations de la programmation défensive est de valider tous les paramètres d'entrée des différents traitements de nos applications.

Elles peuvent:

- Être **malveillantes**
- Entraîner des **erreurs**
- **Les deux à la fois** !

Sur une API REST

Les requêtes

Dans le cas d'une API REST, les différents paramètres d'entrée sont :

- Les paramètres de requêtes : `@RequestParam`
- Les Fragments d'URL : `@PathVariable`
- Les Corps de requêtes : `@RequestBody`
- Les Headers : `@RequestHeader`

NB: On considère en général que sécuriser l'insertion des données et les requêtes suffit à considérer la BDD comme source de confiance.

Hibernate Validattor

Validation automatique

Pour éviter de devoir tester à la main chacun des paramètres, il est possible de déléguer cette tâche à une librairie. La plus connue d'entre elles est **hibernate-validator.**

Par un mécanisme d'annotations, il sera possible d'appliquer des contrôles sur chacun des types d'entrants.

Ajout à une application

Dépendance

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.17.Final</version>
</dependency>
```

Attention : cette dépendance est elle même incluse dans `spring-boot-starter-web`

Contrôler un paramètre

Bean de validation

Pour activer la validation des paramètres de requête et des fragments d'URL, il faut créer un Bean. Sous Spring Boot, l'**auto-configuration** par présence de la librairie dans le classpath s'en charge. **Sinon** :

```
@Bean
public MethodValidationPostProcessor
methodValidationPostProcessor() {
    return new MethodValidationPostProcessor();
}
```

Contrôler un paramètre

Validation d'un Controller

La validation d'un Controller et de ses méthodes se fait en ajoutant l'annotation `@Validated` sur le Controller. Elle déclenche la validation de tous les paramètres pour lesquels un contrôle sera ajouté.

```
@RequestMapping("/test")
@Validated
public class TestController {...}
```

Contrôler un paramètre

Contrôle d'un paramètre

La mise en place de contrôle se fait en ajoutant une annotation au paramètre de la méthode souhaité :

```
@RequestMapping("MonUrl")
public MaClass MaMethode
    (@RequestParam @NotEmpty @Size(min=5) String monParametre){
   ...
}
```

Exemple de contrôles

`@Email`	Le paramètre doit avoir le format d'un email
`@NotEmpty`	La chaine de caractère ne peut pas être vide (différent de `@NotNull`)
`@Size`	La longueur de la chaîne de caractère est contrôlée. Peut contenir les attributs min et max pour spécifier un intervalle de longueur.
`@Range`	La valeur numérique est contenue dans un intervalle de valeurs
`@Pattern`	La chaîne de caractères respecte une expression régulière passée en attribut

https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/package-summary.html

Contrôles = Contraintes

En pleine mutation

Hibernate, le framework dont est issu hibernate-validator, est en pleine restructuration.

Historiquement, il s'appuyait sur des contraintes internes du package **org.hibernate.validator.constraints**. Dorénavant, et afin de se rapprocher des standards, les contrôles à utiliser se trouvent dans **javax.validation.constraints.**

Attention aux imports!

Contrôler un fragment d'url

Contrôle d'un fragment

La mise est identique à celle d'un paramètre de requête.

```
@RequestMapping("/{MonParam}")
public MaClass MaMethode
    (@PathVariable @NotEmpty @Size(min=5) String monParam){
   ...
}
```

Contrôler un header

Contrôle d'un header

La mise est identique à celle d'un paramètre de requête.

```
@RequestMapping("/{MonParam}")
public MaClass MaMethode
    (@RequestParam @NotEmpty @Size(min=5) String monParam){
   ...
}
```

Exception

ConstraintViolationException

Lorsqu'un contrôle n'est pas respecté, une exception de type **javax.validation.ConstraintViolationException** est levée.

Si elle n'est pas catchée, elle peut aboutir en erreur 500 du type :


```
{
    "timestamp": "2019-08-19T07:43:29.904+0000",
    "status": 500,
    "error": "Internal Server Error",
    "message": "test.test: la longueur doit être comprise entre 5 et 2147483647 caractères",
    "path": "tata"
}
```

Améliorer la gestion des exceptions

ControllerAdvice

A priori, un paramètre mal formé ou incorrect devrait plutôt aboutir sur une erreur 400 **Bad Request.**

Il est possible via Spring de définir des méthodes en charge de la récupération des exceptions survenues dans les controllers.

Ces classes sont elle mêmes des Controller, à annoter avec `@ControllerAdvice`

Améliorer la gestion des exceptions

ExceptionHandler

Chacune des exceptions est ensuite gérée dans une méthode annotée avec

```
@ExceptionHandler(value = {MonException.class})
```

Ces méthodes retournent généralement des **ResponseEntity<Object>** pour pouvoir retourner un statut HTTP spécifique et éventuellement un corps de réponse.

Améliorer la gestion des exceptions

Exemple

```
@ExceptionHandler(value = {ConstraintViolationException.class})
protected ResponseEntity<Object> handleConstraintViolationException(
        ConstraintViolationException ex) {
    Map<String, String> errors = new HashMap<>();
    ex.getConstraintViolations().forEach((error) -> {
        String fieldName = error.getPropertyPath().toString();
        String errorMessage = error.getMessage();
        errors.put(fieldName, errorMessage);
    });
    return ResponseEntity.badRequest().body(errors);
}
```

Améliorer la gestion des exceptions

Retour de l'exemple

Body Cookies Headers (4) Test Results Status: 400 Bad Request

Pretty Raw Preview JSON

```
1  {
2      "test.test": "la taille doit être comprise entre 5 et 2147483647"
3  }
```

Contrôler un corps de requête

Dans le controller

Les Corps de requête sont souvent définis dans une classe qui leur est propre. Les contrôles ne s'appliquent pas directement dans le controller, mais dans cette classe.

Seule l'activation des contraintes se fait dans le controller, via l'annotation `@Valid`

```
@RequestMapping(value = "", method = RequestMethod.POST)
public void test(@Valid @RequestBody MaClasse maClasse) {
    ...
}
```

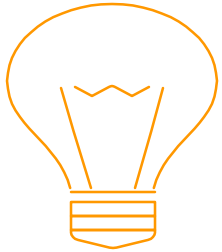
Contrôler un corps de requête

Dans le POJO

Les Contraintes s'appliquent sur chacun des attributs de la classe.

```
public class MaClasse{
    private Integer id;
    @Size(min=1, message = "Nom ne peut être vide")
    private String nom;
    @NotNull(message = "attr2 obligatoire")
    private String attr2;
}
```

Messages des contraintes



Aider le client

Le message d'erreur de chaque contrainte peut être surchargé via l'attribut **message d'une contrainte**.

Il peut être grandement utile de le définir pour un éventuel client.

Améliorer la gestion des exceptions (Pt 2)

MethodArgumentNotValidException

Dans le cadre de la validation du body, les exceptions sont cette fois de type **`org.springframework.web.bind.MethodArgumentNotValidException`.**

Ici Spring renvoie automatiquement un 400 / BadRequest.

Si nécessaire le message peut être remplacé par la paire `@ControllerAdvice` + `@ExceptionHandler`

Améliorer la gestion des exceptions (Pt 2)

Body Cookies Headers (4) Test Results Status: 400 Bad Request Time: 478ms

Pretty Raw Preview JSON


```
{
    "timestamp": "2019-08-19T09:38:13.274+0000",
    "status": 400,
    "error": "Bad Request",
    "errors": [
        {
            "codes": [
                "NotBlank.etudiantDto.nom",
                "NotBlank.nom",
                "NotBlank.java.lang.String",
                "NotBlank"
            ],
            "arguments": [
                {
                    "codes": [
                        "etudiantDto.nom",
                        "nom"
                    ],
                    "arguments": null,
```

Améliorer la gestion des exceptions (Pt 2)

```
@ExceptionHandler(value = {MethodArgumentNotValidException.class})
protected ResponseEntity<Object> handleException(
        MethodArgumentNotValidException ex) {
    Map<String, String> errors = new HashMap<>();
    ex.getBindingResult().getAllErrors().forEach((error) -> {
        String fieldName = ((FieldError) error).getField();
        String errorMessage = error.getDefaultMessage();
        errors.put(fieldName, errorMessage);
    });
    return ResponseEntity.badRequest().body(errors);
}
```

Spring Boot & REST

Journalisation

Journalisation

Définition

Garder une trace d'une séquence d'événements peut être utile pour suivre un procédé, ou comprendre pourquoi une erreur survient.

On appelle Journalisation ou **Logging** le fait d'émettre des messages dans un conteneur dans le but de pouvoir retrouver ces évènements.

Dans l'idéal des mondes, ces traces sont **horodatées** et facilement **exploitables**!

Journalisation

Exemple

```
2019-06-24 14:43:03.474  INFO 6340 --- [           main] org.hibernate.Version                    : HHH000412: Hibernate Core {5.3.7.Final}
2019-06-24 14:43:03.475  INFO 6340 --- [           main] org.hibernate.cfg.Environment            : HHH000206: hibernate.properties not found
2019-06-24 14:43:03.664  INFO 6340 --- [           main] o.hibernate.annotations.common.Version   : HCANN000001: Hibernate Commons Annotations {5.0.4.Final}
2019-06-24 14:43:04.420  INFO 6340 --- [           main] org.hibernate.dialect.Dialect            : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
2019-06-24 14:43:05.614  INFO 6340 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2019-06-24 14:43:06.043  INFO 6340 --- [           main] o.h.h.i.QueryTranslatorFactoryInitiator  : HHH000397: Using ASTQueryTranslatorFactory
2019-06-24 14:43:06.375  INFO 6340 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2019-06-24 14:43:06.413  WARN 6340 --- [           main] aWebConfiguration$JpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may
2019-06-24 14:43:06.624  INFO 6340 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2019-06-24 14:43:06.628  INFO 6340 --- [           main] c.e.studentapi.StudentApiApplication     : Started StudentApiApplication in 6.872 seconds (JVM running for 8.614)
```

Conventions

Solutions de stockage

Dans la grande majorité des applications, ces journaux sont conservés dans un ou plusieurs fichiers texte. Par abus de langage ces fichiers sont souvent eux mêmes appelés **log**.

Il est évidemment possible de les stocker ailleurs :

- Une base de données
- Un mail
- Un appel réseau
- ...

Conventions

Framework

Ce travail de Logging est le plus souvent délégué à une librairie dont c'est le rôle:

- Logback (fourni par défaut avec les dépendances "starter")
- Log4j/Log4j2

Si des éléments de configuration peuvent changer de l'un à l'autre, ils s'appuient tous les deux sur la même terminologie, établie par Log4j.

Niveaux

Criticité du message

L'importance du message est associé à un niveau de Log. Il en existe 5 du moins au plus critique :

- **Trace** : Son usage est très limité.
- **Debug** : Message pouvant être utile pour du debuggage comme des valeurs de paramètres, etc

Niveaux

- **Info** : Message informatif sur l'avance d'un traitement. Peut servir de suivi "grosse maille" dans un déroulé d'algorithme, type : "Step 1 OK", ...

- **Warn** : Un comportement étrange ou inattendu est survenu. Il est tracé pour nécessiter une potentielle intervention plus tard

- **Error** : Une erreur est survenue, empêchant le déroulement normal de l'application.

Terminologie

Logger

Classe appelée pour émettre un message, associé avec un niveau de criticité.

Sa création se fait en utilisant une classe utilitaire de SLF4J, fourni avec Logback : **org.slf4j.LoggerFactory**.

```
public class MaClasse{
    private static Logger LOG =
        LoggerFactory.getLogger(MaClasse.class);
}
```

Terminologie

Hierarchie des logger

Il existe toujours au moins un **rootLogger** en charge de récupérer les traces de l'application.

Lors de l'appel à `getLogger()`, la librairie retourne le logger défini au plus proche en remontant dans la hiérarchie, sinon le rootLogger.

La hiérarchie est construite grâce au nom des classes fully qualified. Ainsi un logger `com.example` est de plus haut niveau qu'un logger `com.example.test`.

Terminologie

Emission

Envoyer un message se fait grâce au méthodes du logger, une par niveau. Trace n'est pas implémenté dans Slf4j.

- `debug(String msg)`
- `info(String msg)`
- `warn(String msg)`
- `error(String msg)`

```
LOG.info("mon Message");
```

Il ne faut jamais logger via System.out.println(...) ni utiliser .printStackTrace()

Ces méthodes ne tracent pas dans le système de log, mais dans la sortie standard.

Terminologie

Appender

Classe utilisée pour envoyer le message à la solution de stockage choisie.

Par défaut, il s'agit d'un **ConsoleAppender** qui ajouter les traces à la console, la sortie standard.

Il en existe autant que de stockages possibles.

Configuration

Fichier de configuration

Logback se configure au travers d'un fichier **xml** devant être dans le classpath / modulepath : **logback.xml** ou **logback-spring.xml**

Y figurent la déclaration des appenders, ainsi que leur association avec les loggers.

Il commence et finit par la balise `<configuration>`

Configuration

Création d'un appender

Appender par défaut pour traçage dans la console.

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
        <pattern>%d [%thread] %-5level %logger{35} - %msg
%n</pattern>
    </encoder>
</appender>
```

Configuration

Il existe plusieurs Appenders pour ajouter des traces à des fichiers, le principal d'entre eux est le **FileAppender**.

```
<appender name="FileDebug" class="ch.qos.logback.core.FileAppender">
    <file><Chemin vers fichier de log></file>
    <append>true</append>
    <encoder>
        <pattern>%d{ISO8601} [%t] %-5level %logger{36} - %msg%n</pattern>
    </encoder>
</appender>
```

Configuration

- `<appender>` démarre la déclaration d'un appender.
  - L'attribut **`name`** est l'équivalent de son identifiant.
  - L'attribut **`class`** correspond à la Classe de l'Appender.

- `<file>` précise le chemin vers le fichier de log.

- `<append>true</append>` spécifie que les prochains messages seront ajoutés au fichier, et qu'il ne sera pas recréé à chaque démarrage.

Configuration

Le `<pattern>` de la balise `<encoder>` spécifie le format de sortie des messages :

- `%d{ISO8601}` : date au format ISO8601
- `[%t]`: Thread d'exécution entre crochets
- `%-5level` : level du message
- `%logger` : le logger ayant émis le message. Correspond à la classe émétrice si bien configuré
- `%msg` : le message émis par le logger
- `%n` : un simple retour à la ligne

NB : Il existe de très nombreux patterns. A consulter au besoin : https://logback.qos.ch/manual/layouts.html

Configuration

Création d'un Logger

Un Logger se déclare comme suit. Sa déclaration contient également le lien avec les Appenders.

```
<logger name="com.example.package" level="info" additivity="false">
    <appender-ref ref="appender1" />
    <appender-ref ref="appender2" />
     ...
</logger>
```

Configuration

- `<logger>` démarre la déclaration d'un logger.
  - L'attribut **`name`** correspond au package auquel il se rapporte.
  - L'attribut **`level`** correspond au niveau de filtre du logger.

- `<appender-ref>` fait le lien avec un appender
  - L'attribut **`ref`** prend la valeur de l'attribut **`name`** d'un **`<appender>`**

Cette mécanique d'association est assez proche de celle des beans Spring en XML.

Les niveaux supérieurs filtrent les inférieurs !

Un Logger définit en level=“warn” n’affiche que les messages de niveau “warn” et “error”.

Configuration V2

AutoConfiguration

Il est possible, via le fichier de properties de configuration, de créer et préciser le niveau des loggers.

L'association avec les appenders y est impossible.

L'utilisation du fichier de configuration Logback reste donc quasi indispensable.

Configuration V2

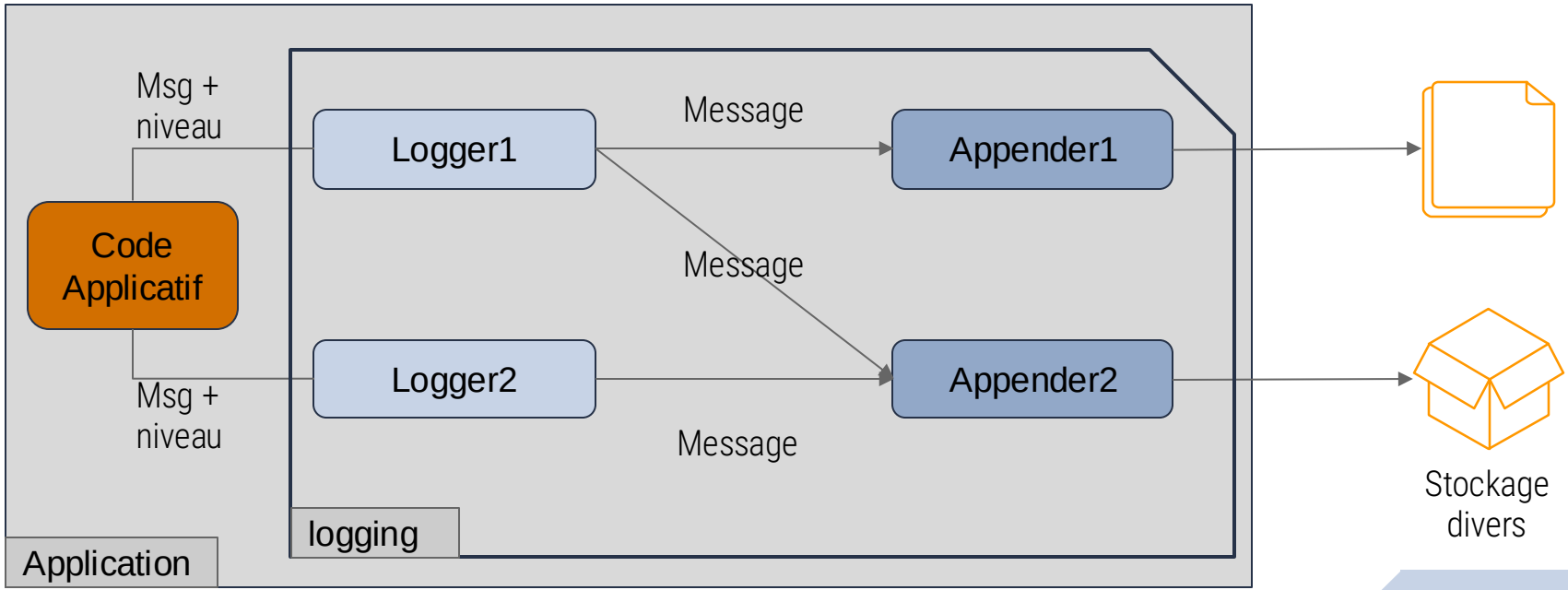
AutoConfiguration

Cette configuration des niveaux **surcharge** celle du fichier xml.

C'est parce que le fichier application.properties (ou .yml) est **externalisable** :

Il peut être passé en argument au démarrage d'une application Spring Boot.

Résumé


Msg +
niveau
Code
Applicatif
Msg +
niveau
Logger1
Logger2
Message
Message
Message
Appender1
Appender2
Stockage
divers
logging
Application