Spring

Constat

Faire une application en JEE “natif” est complexe et long

Code très verbeux

EJB très fastidieux à paramétrer(XML) pour :

- Stocker de la donnée
- Interagir avec les servlets
- Rendre des services

Serveur applicatif lourd pour ces derniers

Constat

Par dessus tout : **très faible tolérance aux erreurs**, très peu explicite!

Un framework pour les gouverner tous

La genèse en 2003 :

Première version de Spring sous Licence Apache

Il est OpenSource!

Objectif n°1 de son créateur à l'époque (Rod Johnson):

Un conteneur "léger"

Il va lui même gérer le cycle de vie des objets nécessaires au fonctionnement d'une application.

Une popularité record

En 2019, ~ 27K github stars: https://github.com/spring-projects/spring-framework

- 4.3.22 (Janvier)

  Version 4.X date de 2013, toujours très utilisée

- 5.1.5 (Fevrier)

  Versions 5.X date de 2017

NB : Spring est devenu polyglotte. Il s'utilise à merveille avec d'autres langages de la JVM kotlin et groovy

Des contraintes de développement

Framework de structure d'application

- Il contraint et "tord" le développement de l'application
- Il peut être compatible avec d'autres frameworks structurant (vertX, Hibernate, …)

Des contraintes de développement

Structure contrainte de l'application : répertoires, patrons de conception, nommage des éléments,...

Les applications Spring présentent de très nombreux points communs qui facilite énormément le passage de l'une à l'autre.

"Rien ne ressemble plus à une application Spring qu'une autre"

Revers de la médaille

Il crée une très forte adhérence entre la structure de l'application et son implémentation.

Il devient difficile de rendre le métier et son traitement agnostique du framework:

Ceci peut poser des problèmes avec certaines tendances actuelles du développement, notamment l'Agilité ou le DDD.

Une galaxie de services

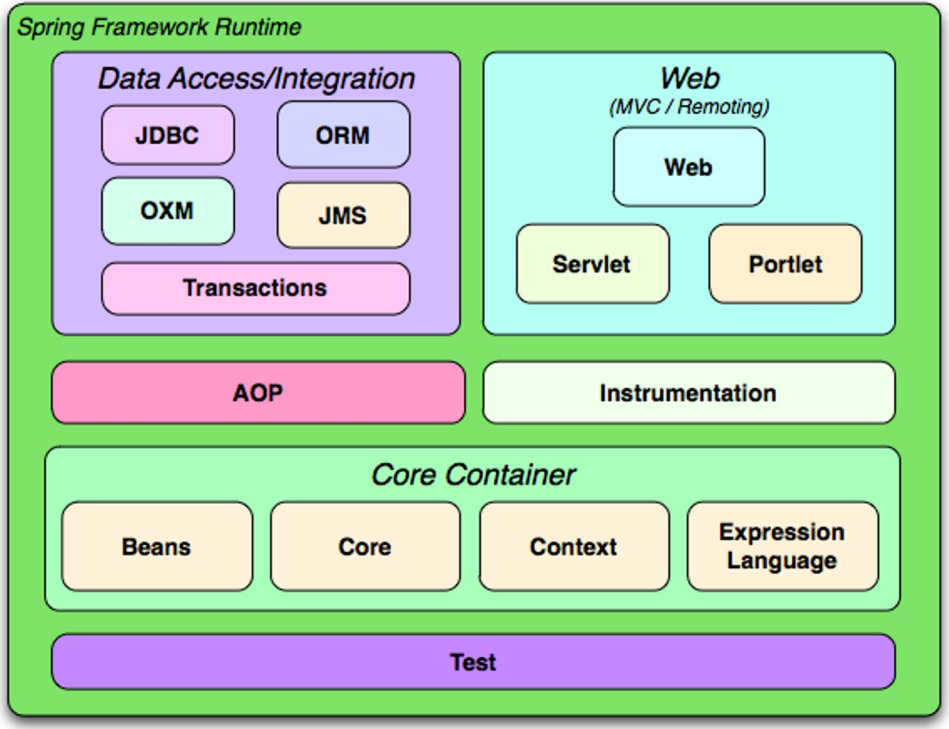
“Il y a une solution spring pour à peu près tout”

Depuis sa création, Spring a énormément grandi.

Parti d’une volonté de simplification, il est devenu une gigantesque boîte à outils facilitant tout un tas d’API du JEE : base de données, requêtes HTTP, ...

Une galaxie de services

Principaux modules



Une découpe en modules

Core: Noyau du framework. Contient les classes utilisées dans tous les autres modules, ainsi que le conteneur léger et sa mécanique.

Web: module permettant l'ouverture des application Spring sur le Web. Il contient notamment Spring MVC, qui va s'interfacer entre le métier et la vue à proprement parler. Il s'accorde avec de nombreuses solutions de rendu serveur (thymeleaf, JSF, JSP,...)

Une découpe en modules

Data Access: Regroupe les modules d'accès aux données d'une application, base de données en tête. JDBC propose une implémentation très proche du SQL, alors qu'ORM va plutôt faire le lien entre Objets et modèle relationnel.

Test: Intègre certaines mécaniques de Spring à Junit : contexte, Runner spécifique, etc.

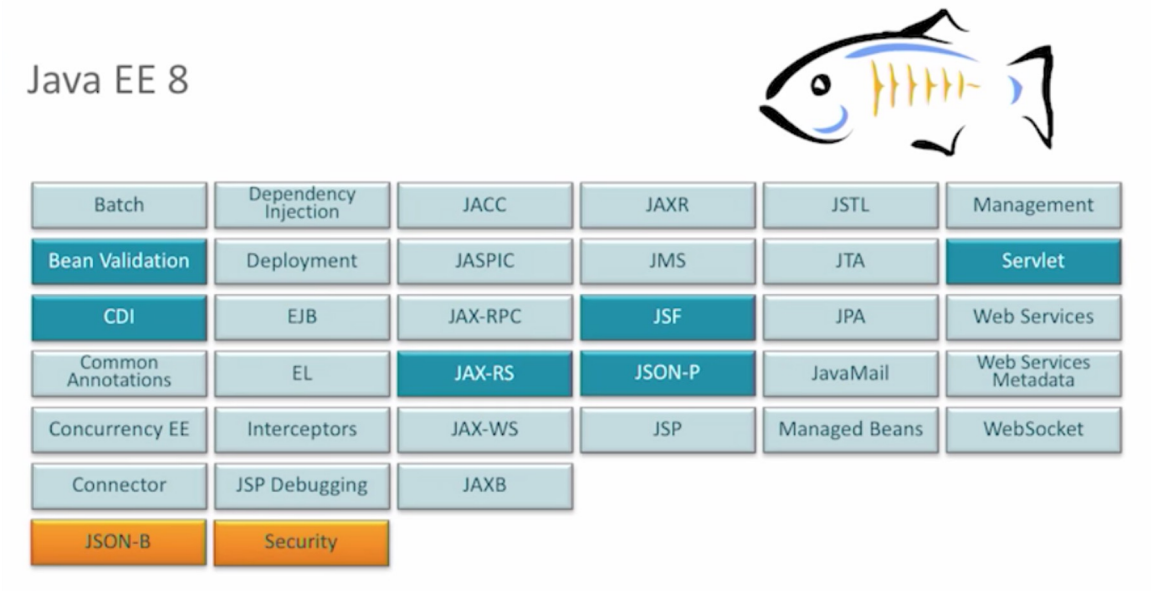
Une découpe en modules

AOP: Solution spring pour la programmation orientée Aspect. S'intègre et étend certaines fonctionnalités d'AspectJ.

Instrumentation: Ensemble de composants permettant le monitoring et l'exploitation des applications. Ils peuvent servir à la fois dans l'alerting , le diagnostic des erreurs ou la journalisation des événements

Une galaxie de services

Un parallèle avec le JEE



Un éco-système très profond

Vous avez dit Galaxie?

- Spring Batch
- Spring Security
- Spring Data
- Spring Ldap
- Spring Web Services
- ...

Le strict minimum

Pour pouvoir créer et utiliser des beans

- Spring-core
- Spring-beans
- Spring-context

Spring

Mes premiers design pattern

Cycle de vie

En JEE, le serveur d'application va gérer le cycle de vie des EJB, leur disponibilité et leur portée.

C'est ce rôle que prend Spring dans une application.

Le contrat de service

Couplage fort...

La classe concrète est connue de l'appelant :

```
ArrayList<String> maListe = new ArrayList<>();
```

Il a accès à toutes les méthodes de la classe ArrayList et connaît donc les détails de son implémentation.

Le contrat de service

… vs Couplage Faible (voire lache)

La classe concrète est inconnue de l'appelant :

```
List<String> maListe = new ArrayList<>();
```

Il ne connaît que l'interface et fait confiance à l'implémentation.

On parle de **Contrat de Service**.

Spring Context

Ensemble des objets connus et managés par Spring.

Sorte de banque de classes que le framework peut utiliser dans l'application.

Tout est une histoire de contexte

Par défaut, configuré par le fichier “applicationContext.xml”, à placer dans le ClassPath de l’application, généralement dans un répertoire dédié aux ressources de configuration.

Il peut être remplacé par de la configuration Java et des annotations.

Correspond à l’interface **applicationContext**

Spring Bean

Objet managé par Spring, contenu dans le contexte du framework.

Les beans possèdent un nom, un identifiant et un type.

Les beans

L’épine dorsale de l’application

Tout s’articule avec et autour d’eux.

Une bonne pratique veut qu’ils soient des **implémentations d’interface** : seul le résultat compte.

Exemple de beans :

Liste de valeurs, connexion à une base de données, classe utilitaire contenant du code métier, ...

Les beans

Exemple de fichier applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">


 <bean name="testBean" class="com.test.Bean">

   <property name="testProperty" value="Hello world"/>

 </bean>

</beans>
```

Déclaration d'un bean

Un bean spring peut être déclarée de plusieurs manières :

- Dans un **fichier xml** de configuration de type applicationContext.xml
- Dans sa propre classe, par un mécanisme d'annotation : `@Component`, `@Service, @Controller`...
- Dans une classe dite de Configuration annotée `@Configuration`

NB: Les annotations de classes type `@Component` ne fonctionnent que dans des packages "scannés" par spring. Ils se spécifient grâce au "component-scan"

Déclaration d'un bean

Dans un fichier de **configuration xml**.

```
<bean name="testBean" class="com.test.Bean">
  <property name="testProperty" value="Hello!"/>
</bean>
<bean name="testString" class="java.lang.String">
  <constructor-arg value="test"></constructor-arg>
</bean>
```

Déclaration d'un bean

Dans un fichier de **configuration Java**

```
@Bean
public String testString() {
    return "test";
}
@Bean
public Bean testBean() {
    Bean monBean= new Bean();
    monBean.setTestProperty("Hello!");
    return monBean;
}
```

Déclaration d'un bean

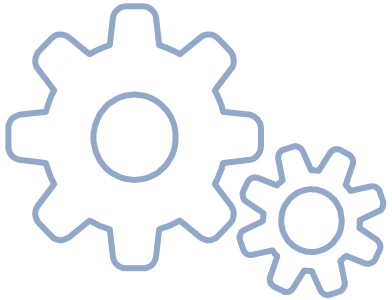
Via annotation sur la classe:

```
@Component
public class JavaComponent {
  /**
  * Crazy awesome stuff
  */
}
```

Ici, Spring va créer et mettre dans son contexte une instance de JavaCompoment. Toute la puissance de Spring est accessible

dans cette classe

Spring ne voit pas tout



Un objet instancié au sein d'une méthode n'est *par défaut* pas connu de Spring.

Il ne peut bénéficier d'aucune des fonctionnalités de Spring.

Scan de classes

Les yeux de Spring

Pour qu'une classe annotée par `@Component`, ou par un autre "Stéréotype" soit connue de Spring, et qu'il en ajoute une instance dans son contexte, il faut préciser au framework où regarder.

`@ComponentScan("mon.package")` va préciser à Spring ou se trouvent les classes annotées qu'il doit ajouter au contexte. Elle doit être ajoutée sur une classe de configuration.

L'équivalent XML de cette configuration est :

```
<context:component-scan base-package="org.example"/>
```

Inversion de contrôle (IOC)

Pierre angulaire de Spring

C’est un principe de conception d’une application dans lequel le contrôle des objets et de leurs dépendances est transféré à une autre entité.

Chez Spring, il s’appuie en partie sur le Design Pattern Factory vu précédemment.

Inversion de contrôle (IOC)

The Hollywood Principle

Inversion de contrôle (IOC)

Dépendance

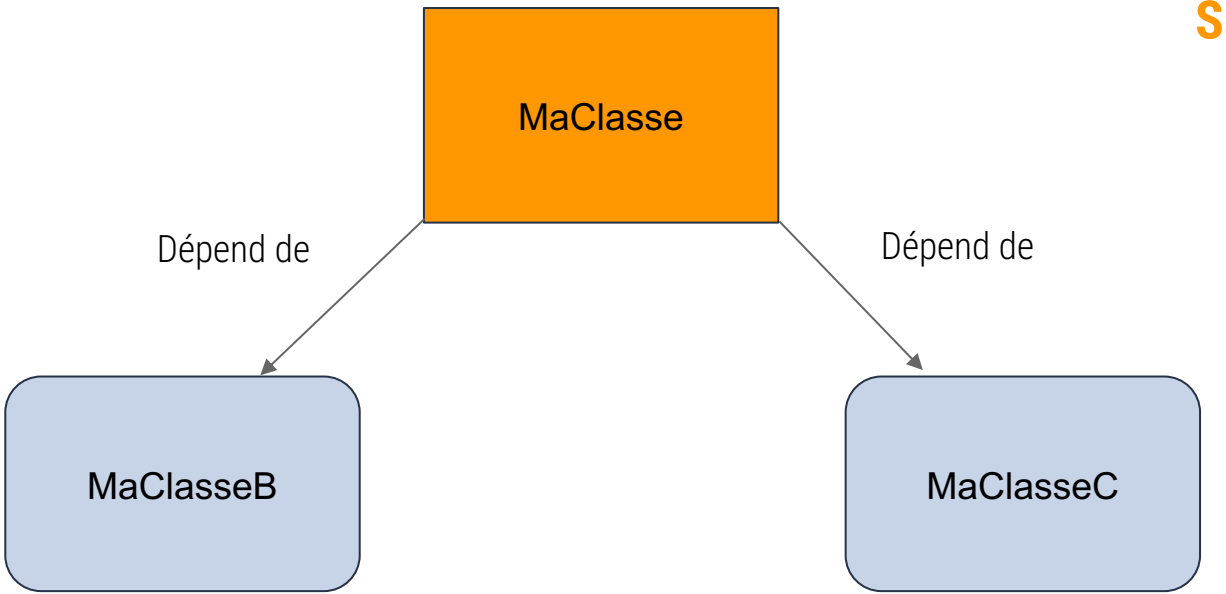
Soit MaClasse, MaClasseB et MaClasseC trois classes reliées comme ci dessous :

```
public class MaClasse {

    private MaClasseB classeB;
    private MaClasseC classeC;
}
```

Inversion de contrôle (IOC)

Sans IOC



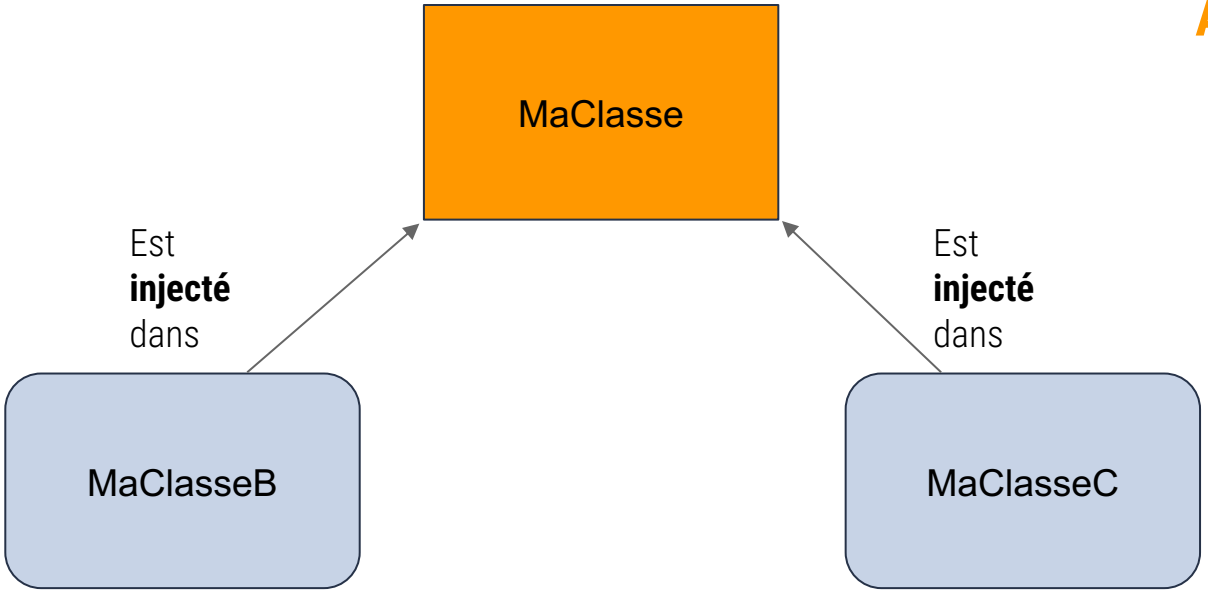
Inversion de contrôle (IOC)

En conception classique, l'instanciation de MaClasse se ferait ainsi:

```
MaClasse maClasse = new MaClasse();
maClasse.setClasseB(new MaClasseB());
maClasse.setClasseC(new MaClasseC());
```

Inversion de contrôle (IOC)

Avec IOC



Inversion de contrôle (IOC)

Avec IOC, l'instanciation de MaClasse par Spring pourrait prendre la forme suivante :

```
MaClasse maClasse =
WebApplicationContextUtils.getWebApplicationContext(...).getBean
("maClasse")
```

`WebApplicationContextUtils.getWebApplicationContext(...)` étant ici l'appel au contexte spring, "référentiel" de tous les Beans

Injection de dépendances

Il s’agit du pattern d’implémentation de l’inversion de contrôle choisi par Spring.

Dans ce pattern, les associations entre les objets sont faites par un “Assembleur” plutôt que par les objets eux mêmes.

C’est ce rôle que tiennent les classes internes de Spring.

Le cycle de vie des beans

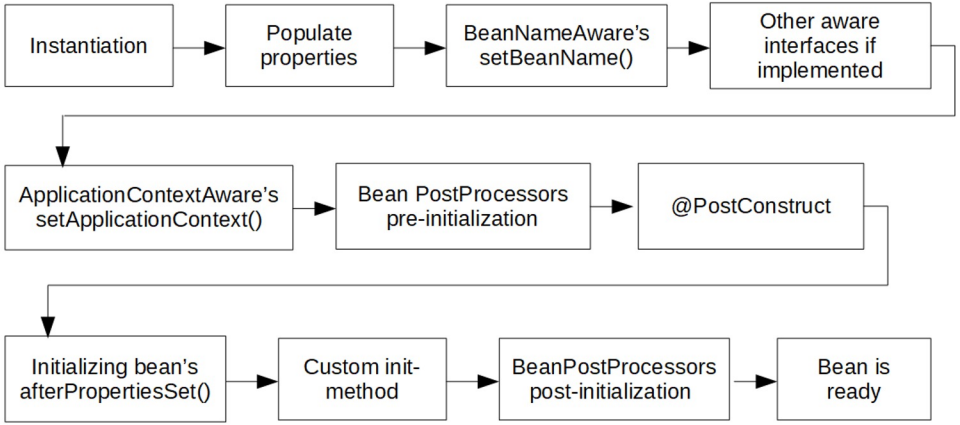
“A matter of Life and Death”

De leur création à leur destruction par Spring, tous les beans gérés par ce dernier suivent un Cycle de Vie.

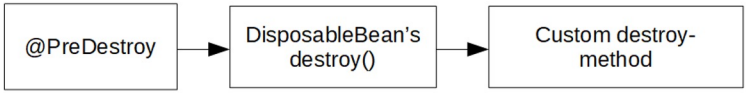
Il s’agit d’un enchaînement d’étapes composée d’appels de méthodes, internes à Spring ou pas, qui vont permettre :

- La création de l’Objet
- L’ajout de ses attributs
- La déclaration en tant que bean dans le contexte Spring
- ...

Le cycle de vie des beans



Call back method flow after bean instantiation



Call back method flow for disposing bean

Chaînage des injections manuel

L'équivalent en configuration Java de l'exemple précédent est le suivant :

```
@Bean
public Bean testBean(@Autowired Bean2 injectedBean) {
    Bean bean = new Bean();
    bean.setTestProperty(injectedBean);
    return bean;
}
```

L'annotation **@Autowired** spécifie que cet objet doit venir du contexte Spring.

Injection automatique

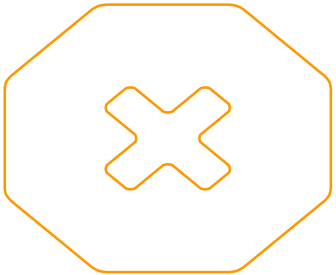
L’annotation **@Autowired** ouvre la voie de l’injection dite “automatique”.

Si un élément (un paramètre de constructeur, un paramètre de Setter, un attribut de classe) est annoté avec, Spring va tenter de trouver une correspondance possible avec l’ensemble des Beans dans son Context.

Injection automatique

Historiquement, l'annotation @Autowired se plaçait sur les attributs d'une classe.

```
@Autowired
private Bean bean;
```



Désormais, il est recommandé de l'utiliser au sein d'un constructeur, ou d'un Setter.

Injection automatique

Mécanisme d'injection automatique

La résolution des dépendances à injecter se fait suivant 2 mécaniques :

- **byName:** Spring va chercher un bean dont l'identifiant est le même que le nom de la propriété ou du paramètre que l'on souhaite injecter
- **byType:** Cette fois c'est le type de l'objet que Spring va chercher à faire coïncider à une bean existant et connu. Lorsque rien n'

Si aucune correspondance n'est trouvée par ces 2 mécaniques, spring abandonne la résolution et lève une exception.

Mécanisme d'injection automatique

`@Autowired(required = false)` permet de ne pas lever d'exception si la résolution de l'injection échoue.

Les objets non résolus sont laissés à **null**. D'autres exception sont donc fort probables lors de l'appel à ces éléments.

Injection automatique

Pré-requis

`@Autowired` ne fonctionne que dans les classes elles-même connues du contexte de Spring.

Elle n'a aucun effet dans par exemple :

- Les Pojos
- Les tests Junit

Injection automatique

Forçage de l'injection

`@Qualifier` permet de forcer la mécanique d'injection automatique en forçant l'usage du nom.

Elle sert à résoudre un problème spécifique :

Si plusieurs beans peuvent correspondre au critère d'injection (**ie**: qu'ils ont tous la bonne classe), comment Spring peut il choisir?

Injection automatique

```
@Autowired
@Qualifier("monbean")
private MonBean monBean;
```

Le code ci dessous force l'injection d'un bean de type MonBean, et nommé "monbean".

Le paramètre de l'annotation correspond au nom du bean:

- Défini dans l'attribut "name" du xml
- Défini dans le nom de la méthode en configuration Java

Injection automatique

```
@Component
@Qualifier("monbean")
Public MaClasse {}
```

Posée sur une classe, elle même annotée par spring, l'annotation permet de définir le nom d'un Bean défini par annotation.

Initialisation tardive des beans

Démarrage très long

L'un des reproches formulé au JEE est sa lenteur au démarrage.

La mécanique des beans de Spring accentue encore cet effet:

Par défaut, tous les beans sont créés au démarrage de l'application.

Initialisation tardive des beans

Cache misère

Ajouter l'annotation **@Lazy** avec @Autowired permet de n'instancier le bean demandé que lorsqu'il est utilisé.

Cette mécanique ne fonctionne toutefois qu'avec les beans qui ne sont pas créés directement par Spring, et donc uniquement le code purement applicatif du développeur.

Scope d’un bean

Un seul ou plusieurs beans à la fois

Un bean est associé à une stratégie d’instanciation appelée **Scope.** Plusieurs valeurs de ce scope sont possibles :

- **Singleton:** C’est le scope par défaut. Une seule instance du bean est créée par Contexte Spring.
- **Prototype:** Une instance du bean est créée pour chaque référence du bean.

Scope d’un bean

- **Request:** Une instance du bean est créée pour chaque requête HTTP. N’est fonctionnel que dans un contexte Web

- **Session:** Une instance du bean est créée pour chaque session HTTP. N’est fonctionnel que dans un contexte Web

- **GlobalSession:** Une instance du bean est créée pour chaque session HTTP globale. N’est fonctionnel que dans un contexte Web

Scope d’un bean

En configuration Java, ce scope est spécifié via l’annotation @Scope

```
@Bean
@Scope("singleton")
public Bean testBean() {}
```

Scope d’un bean

En configuration XML, ce scope est spécifié via l’attribut scope

```
<bean id="bean" name="testBean" class="com.test.Bean" scope="singleton">
```

Spring

Spring Boot, sa philosophie

Constat

Spring c’est bien!

Il facilite énormément la mise en place de mécaniques courantes dans le monde du web : traitement de requêtes HTTP, connexion à une base de données, gestion des logs, etc.

Il est bien plus léger et moins contraignant que ne l’est la stack JEE par défaut.

La programmation par interface facilite l’adaptabilité.

Constat

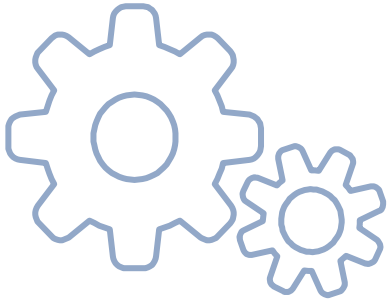
Mais pas top!

“Springifier” une application nécessite beaucoup de configuration.

Il y a par ailleurs une certaine répétitivité dans la création de cette configuration :

- Récupération des dépendances
- Vérification des versions entre elles
- Créations des beans classiques (connexion à une base, gestion de l’internationalisation, ...)

“Un bon développeur est un développeur fainéant”



Faire deux fois les mêmes gestes n’est pas gratifiant. Le faire 50 fois encore moins, et fait perdre du temps!

Principes de base

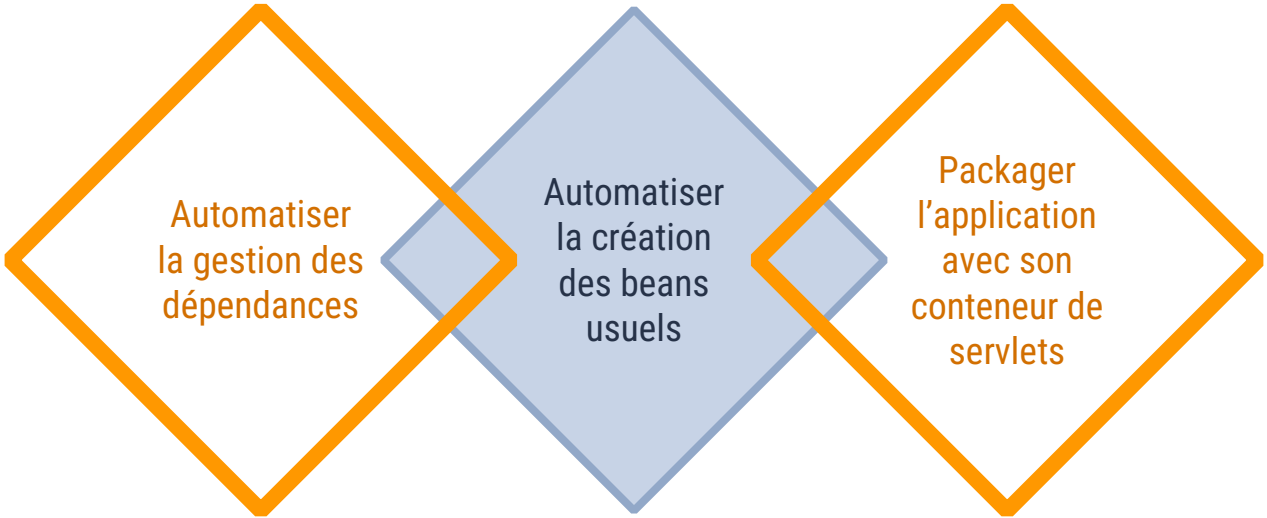
Automatiser ce qui ne produit pas de valeur

Spring Boot vise à remplacer par de l'automatisme tout ce qui n'est pas réellement lié à la production de valeur d'une application.

Supposons que l'on souhaite créer une application de gestion des réservations de salles.

La vraie plus value est bien dans la réservation de salles, pas dans la configuration des sempiternels mêmes mécanismes!

Une conjonction d'idées



Automatiser la gestion des dépendances

Spring boot propose des “**super dépendances**” en regroupant plusieurs autres souvent utilisées ensemble.

Par exemple, Spring Data est souvent utilisé avec Spring Transaction et un gestionnaire de transactions.

Malheureusement, toutes les versions ne sont pas compatibles entre elles, et trouver la bonne combinaison peut virer au casse tête.

Automatiser la gestion des dépendances

Pour y remédier, Spring boot propose une dépendance “spring-boot-starter-data-jpa” qui regroupe les 3, ainsi que d’autres.

Cette dépendance est disponible sous plusieurs versions, embarquant elles mêmes des combinaisons différentes des librairies la composant.

Ainsi, quand une nouvelle version majeure de Spring Transaction paraît, une nouvelle version majeure de spring-boot-starter-data-jpa paraît également.

Automatiser la gestion des dépendances

Choix de standards

Bien évidemment, toutes les librairies ne sont pas disponibles dans ces “méta-dépendances”.

Les équipes de Spring ont du faire des choix, qu’ils ont jugé standard.

Dépendances de Spring Boot

Pom parent

Une partie des dépendances est amenée par héritage du pom parent Spring Boot

```
<parent>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-parent</artifactId>
 <version>2.0.5.RELEASE</version>
</parent>
```

Dépendances de Spring Boot

Plugin

Un plugin maven permet d'intéragir avec spring Boot pour le démarrage, les dépendances, et certaines options de packaging.

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

Dépendances de Spring Boot

Dépendances

Les “super dépendances” s’ajoutent de manière classique :

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

Dépendances de Spring Boot

Principales dépendances

`spring-boot-starter-web:` Utile pour exposer sur le web une application : API Rest, Spring MVC, etc.

`spring-boot-starter-test:` Packaging pour les tests unitaires et d'intégration : Junit, Mockito, etc.

`spring-boot-jdbc-starter:` Utilitaire de connexion à une base de données

Dépendances de Spring Boot

Principales dépendances

`spring-boot-starter-security:` Utile pour sécuriser l'accès à une application, basé sur Spring security et toutes ces dépendances usuelles

`spring-boot-actuator:` Ensemble de solutions de monitoring pré intégrées aux applications et fournies par Spring

. . . . La liste est très longue!

Automatiser la création des beans usuels

Éviter la répétitivité

Beaucoup d'éléments de configuration sont communs entre des applications dont le coeur de métier est différent :

- Récupération d'URL
- Configuration d'un serveur SMTP
- Sérialisation d'objets Json
- ...

Automatiser la création des beans usuels

Auto-configuration

Spring boot propose au développeur de s'affranchir de cette phase de configuration, répétitive, souvent à base de copier coller d'exemples ou d'applications existantes.

Ainsi le développeur est focalisé sur la **vraie valeur de son produit : le métier**

Ce mécanisme est nommé **Auto Configuration**

Auto Configuration

Fonctionnement

Pour savoir quels beans doivent être créés par l'application, et ajoutés au contexte Spring, Spring Boot se base, entre autre, sur un fichier de configuration principal.

Ce fichier, peut, par défaut être nommé de deux manières :

- **Application.yml** : Structure de fichier yml, où l'indentation fait la structure

  OU

- **Application.properties**: Structure de fichier properties

Auto Configuration

Exemple application.yml

```
spring:
 datasource:
   url: jdbc:h2:mem:testdb
   password:
   username: test
   driver-class-name: org.h2.Driver
```

Auto Configuration

Exemple application.properties

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.password=
spring.datasource.username=test
spring.datasource.driver-class-name=org.h2.Driver
```

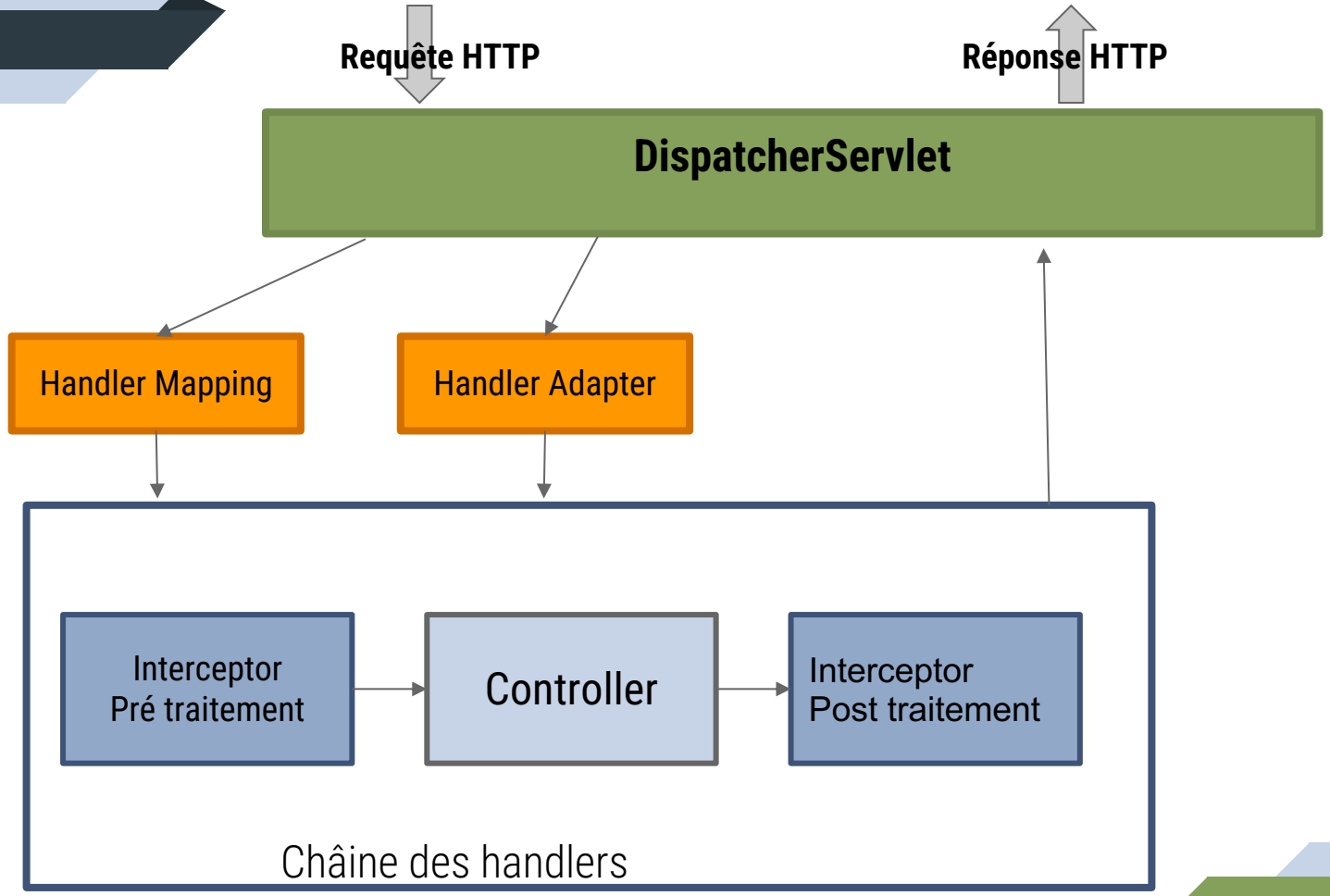
Une Main Class

Le démarrage d'une application Spring Boot est conditionné à la présence d'une classe dite "**Main Class**".

Cette classe **doit** contenir une méthode avec cette méthode:

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

Ce sont les annotations posées sur cette classe qui vont conditionner la création du contexte Spring, de l'auto configuration etc.


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`.

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 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`