Spring

Plan

- Framework, librairies et gestionnaire de dépendances
- Spring, qu'est ce que c'est?
- Mes premiers design patterns
- Spring Boot, sa philosophie

Spring

Frameworks, librairies et gestionnaire de dépendances

Librairies externes

Java n'est pas un langage neuf (1995) : il traîne avec lui beaucoup d'historique!

Dans l'informatique, il est très peu probable que vous soyez les premiers à rencontrer un problème : quelqu'un d'autre s'est probablement déjà cassé les dents dessus par le passé.

Tous les niveaux de difficultés sont concernés (lire un fichier, trier des éléments, communiquer avec une machine distante, imprimer des documents, etc)

Librairies externes

Peut être est-il possible de s’appuyer sur leurs réalisations?

Librairies externes

Comment utiliser d'autres éléments que je n'ai pas développé ? 3 solutions pas exclusives

- Les imports
- Le Classpath
- Depuis java 9 : les modules

Import <Package>.<Class>

Mécanisme élémentaire!

L'instruction **Import** permet d'ajouter la visibilité d'une autre classe à la classe courante.

Se trouve en début de classe et utilise la syntaxe :

Import mon.package.MaClasse

Les IDE modernes cachent cette mécanique en la gérant pour les développeurs

Classpath

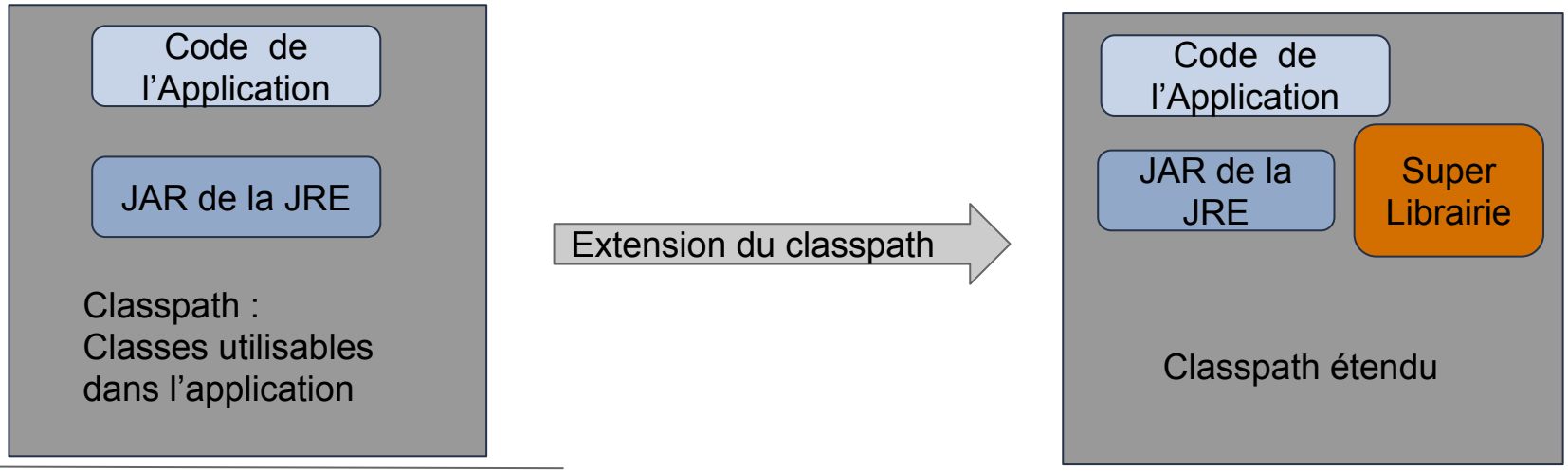
Classpath : Solution Java permettant de localiser l'ensemble des classes et des **JAR** accessibles au sein d'une application Java.

Tout ce qui se trouve dans le classpath peut être importé dans une application et utilisé à l'exécution.

Par défaut, il contient l'ensemble des librairies fournies avec la JRE (exécution et non compilation), ainsi que le répertoire courant d'exécution de l'application.

Classpath

Il peut être étendu et modifié pour ajouter d'autres éléments via des arguments au démarrage de la JVM : **java -cp repertoire1;repertoire2;...**

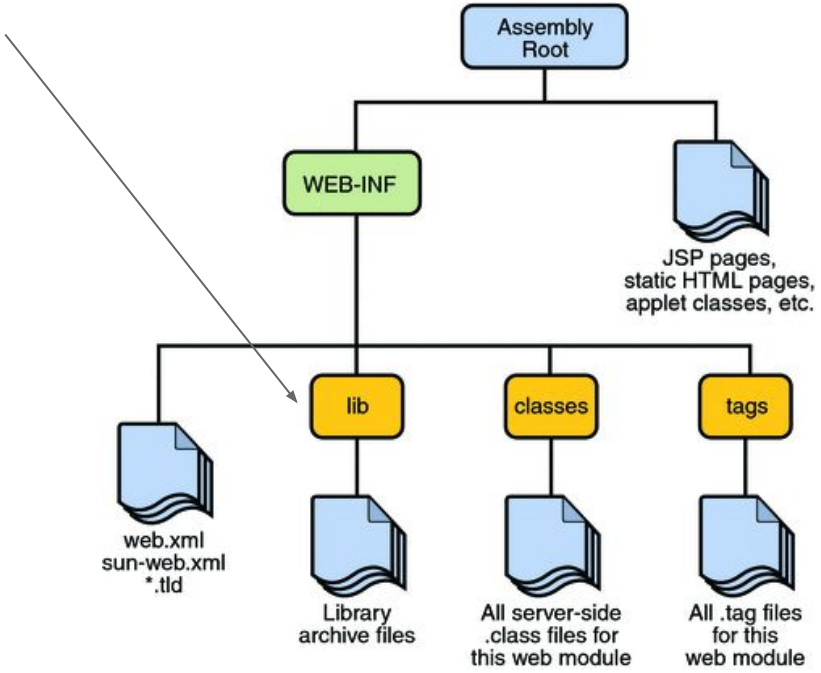


Super Librairie

Élements inaccessibles pour la JVM

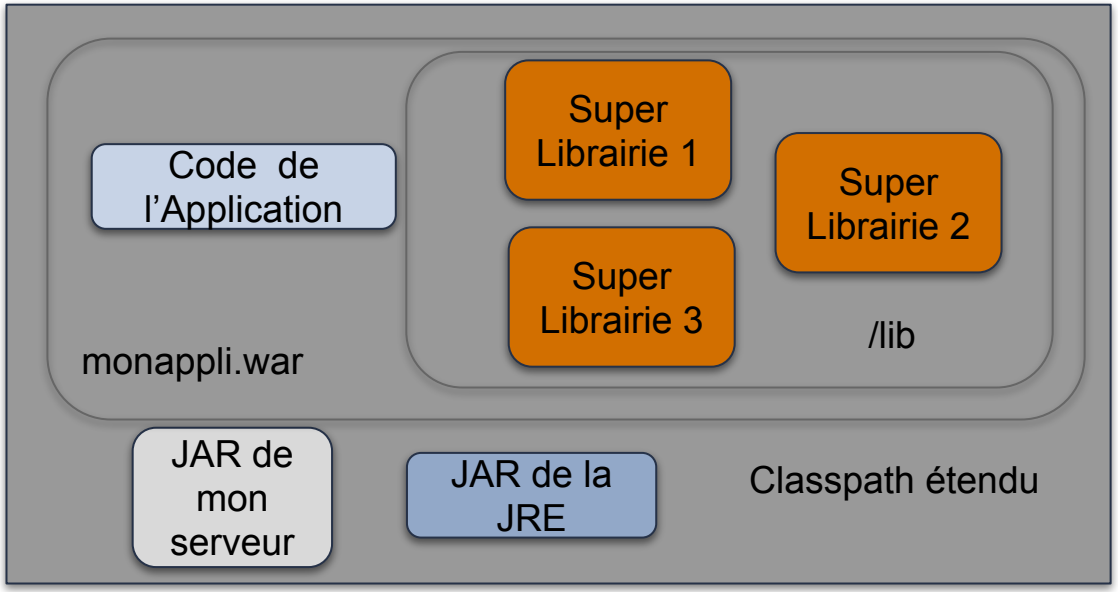
Dans le contexte WEB

Les librairies externes se trouvent au sein du WAR créé dans la phase de compilation, dans le répertoire **/lib**



Le serveur web se charge de la construction et de l'extension du classpath.

Il y ajoute d'ailleurs les librairies nécessaires à son fonctionnement.



Le ClassLoader

C'est le mécanisme de chargement des classes au sein de l'application :

- Il va les chercher dans le classpath lors de l'instanciation ou de l'appel à une méthode statique
- Il parcourt linéairement le classpath, et s'arrête à la première correspondance

  Ceci implique qu'il existe une **notion d'ordre très importante dans le classpath**

- Aucun contrôle n'étant fait au démarrage, il est possible qu'une classe soit manquante à l'exécution car non embarquée à la compilation

Dependency Hell!

L'utilisation d'une librairie externe est aussi appelée **Dépendance** à cette librairie.

Certaines dépendances amènent deux versions différentes de la même classe.

Le ClassLoader en "choisit" une , qui n'est peut être pas la bonne !

On appelle ce phénomène le **dependency Hell.**

C'est une vraie difficulté dans la conception d'une application, qui peut nécessiter de nombreuses heures de correction!

Modules

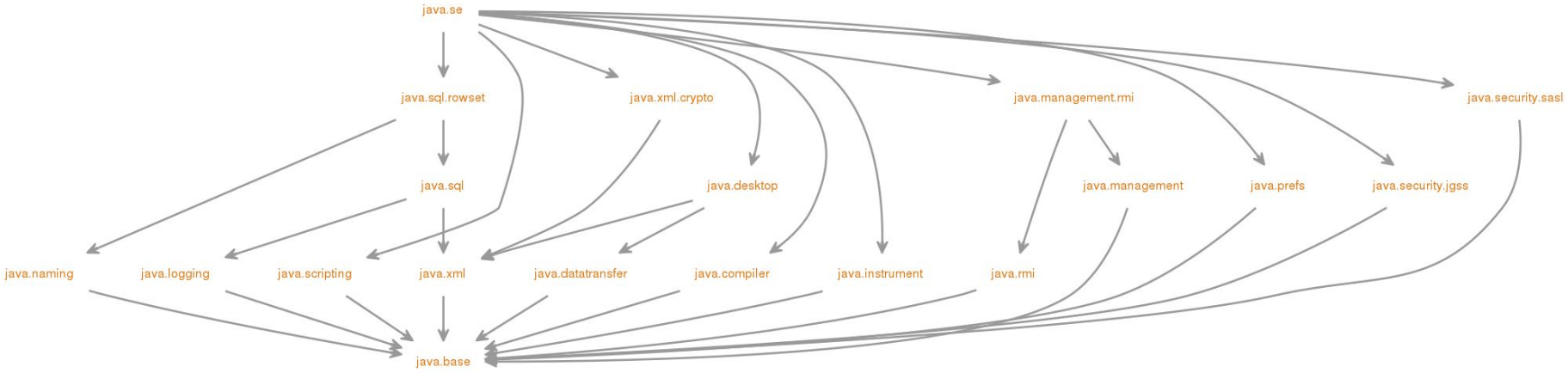
Nouvelle feature de Java 9 : **refonte du classpath**. Anciennement appelé projet Jigsaw.

Ils répondent en partie au problème soulevés par le classpath et le dependency Hell :

- Contrôle des librairies présentes au démarrage de l'application
- Hiérarchie dans les dépendances, permettant une priorisation
- Meilleure visibilité des éléments accessibles depuis une librairie : seul ce qui est explicitement exporté est visible.
- Seul ce qui est nécessaire est embarqué : réduction de la taille des applications

Modules

Voici la nouvelle architecture de Java avec l'approche modulaire : un arbre!



Modules

Ils présentent une nouvelles contrainte pour les applications existantes :

Le code existant peut nécessiter une réécriture ! Il n'est plus permis d'utiliser des éléments non exportés. Ceci a entraîné la réécriture complète de librairies connues et a même touché la JRE elle même.

La majorité des grandes librairies sont maintenant disponibles en modules.

Il vaut mieux prendre en compte cet aspect rapidement dans le développement ou choisir de l'ignorer totalement. Attention : cette feature est fortement poussée par Java et risque de prendre de l'ampleur

Gestionnaire de dépendances

Pour aider le développeur dans la gestion des librairies externes, il existe des outils dédiés :

Les gestionnaires de dépendances.

Il en existe actuellement 2 principaux pour le monde Java : **Gradle** et **Maven**.

Tous les deux s'appuient sur un fichier de description souvent à la racine des sources:

- build.gradle pour Gradle
- pom.xml pour Maven

Maven

http://maven.apache.org

Outil OpenSource développé par la fondation Apache.

Il sert notamment à l'automatisation de certaines tâches :

- Exécution de tests, génération de la documentation,...
- Téléchargement des librairies externes utilisées dans une application
- Construction du livrable de l'application contenant tous les éléments nécessaires à son déploiement sur un conteneur de servlets ou un serveur applicatif

Fonctionnement général

C’est une application externe à installer indépendamment. Elles nécessite la présence d’un JDK et la configuration de certaines variables d’environnement.

Dernière version à date : 3.6.0

Elle s'exécute en ligne de commande: **mvn** <option> dans un répertoire contenant un fichier pom.xml décrivant les opérations.

Elle peut aussi être intégrée à l’IDE.

Terminologie

- **pom ou pom.xml** : Fichier décrivant le comportement que maven doit suivre lors de son exécution. Par défaut à la racine du projet
- **repository** : répertoire contenant les dépendances déjà téléchargées dans le passé par maven.
- **settings ou settings.xml** : Fichier contenant les configuration spécifiques d'un utilisateur (authentification, localisation du repository, etc). Par défaut dans le répertoire personnel de l'utilisateur
- **lifecycle** : Cycle de vie maven. Ensemble des opérations effectuées par l'outil

Build lifecycle en détail

3 Builds lifecycle existants :

- **clean** : nettoie les sources de tous les éléments générés par maven, principalement créés dans un répertoire **target** à la racine
- **site** : génère un site utilisable par maven pour le projet (peu utilisé)
- **default** : permet la construction et le déploiement de l'application et de ses binaires

Ces lifecycles sont eux mêmes divisés en phases, variable d'un build à l'autre.

Phases du build default

6 principales phases le composent :

- **validate** : Contrôle structurel du pom et du projet
- **compile** : Compilation des sources en binaires
- **test** : Exécution des tests embarqués dans l'application
- **package** : Création du livrable a partir des éléments compilés
- **install** : Installation du livrable dans le repository local
- **deploy** : Envoi du livrable sur un repository distant

D'autres phases mineures sont également présentes (plus de 20 au total)

Phases du build default

Par défaut, chacune des phases est bloquante pour la suivante. Ainsi, si le projet ne compile pas, ou que les tests échouent, le livrable ne sera pas créé.

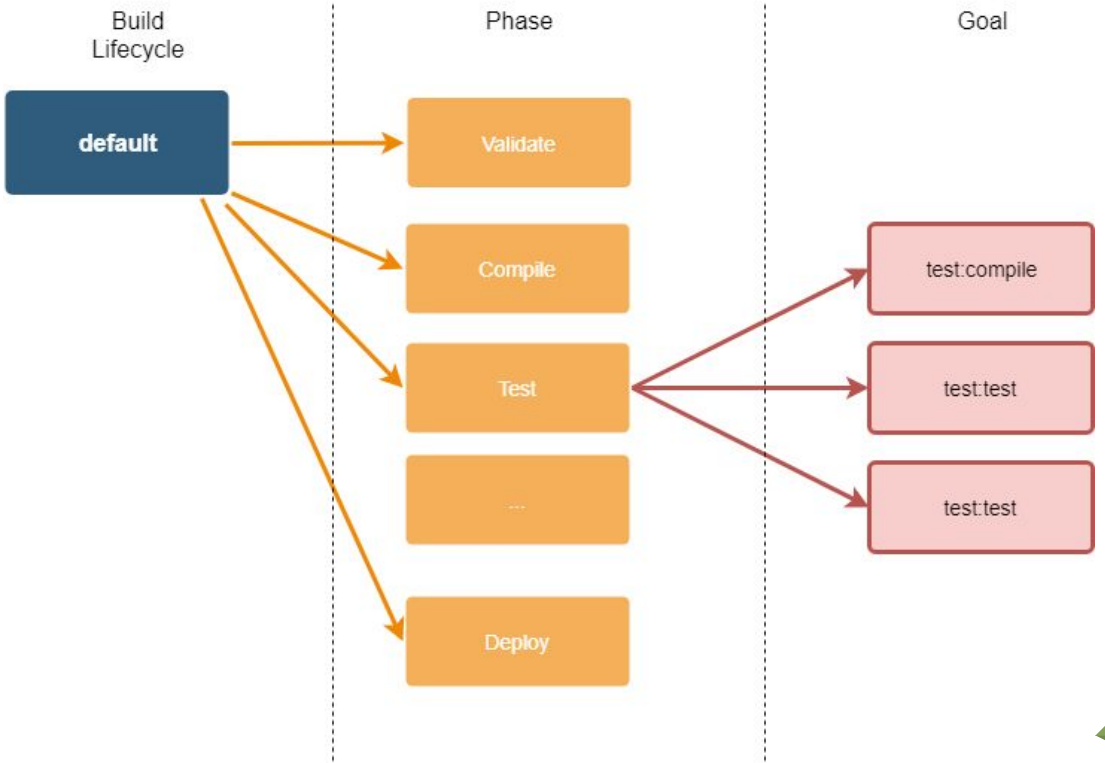
Phases du build default

Dans les faits, la granularité d'une phase est encore plus précise. Chacune d'entre elles se découpent en **goal** qui s'enchaînent de la même manière que les phases.

Certains de ces goals peuvent être modifiés ou ajoutés via la configuration décrite dans le fichier pom.xml et les **plugins**. Un certain nombre d'entre eux sont présents par défauts dans maven.

Par exemple la phase "compile" va s'appuyer sur le plugin "compiler" et le goal "compile". Notation : **plugin:goal**

Lifecycle : résumé



Le fichier pom.xml

Fichier .xml régi par une xsd : un fichier de contrôle de structure. Sa rédaction doit être très rigoureuse! Syntaxe HTML pour les commentaires.

En cas d'erreur, la commande mvn ne peut être exécutée.

Sa structure est découpée en plusieurs blocs/balises xml, chacun d'eux étant chargé d'apporter des éléments sur le comportement de maven :

- nom du projet
- dépendances à ajouter
- tests à ignorer
- scripts à jouer,
- …

Le fichier pom.xml

Les plugins (liés aux goals) offrent énormément de possibilités :

“Un grand pouvoir implique de grandes responsabilités”

Il est possible d’utiliser maven pour des comportements qui n’étaient pas sa cible initiale: démarrage de serveurs, émissions de requêts HTTP, etc.

Attention : il est préférable d’utiliser un outil spécifique et dédié pour ce genre de tâches.

Le fichier pom.xml

Principales balises du fichier pom.xml:

`<groupId>` : Identifiant de regroupement du projet, servant, entre autre, à son rangement dans un repository maven. Utiliser la syntaxe de type “package”

`<artifactId>` : Nom du projet, espaces impossibles. Préférer le kebab ou le kamel case

`<packaging>` : Format du livrable : pom (pour des hiérarchies de projets), jar, war, etc.

`<version>` : Version du projet. Le projet évoluant dans le temps, ce chiffre augmentera au fil de son développement

`<name>` : Nom du projet plus explicite, sans limite de saisie

`<description>` : Bref descriptif du projet

Le fichier pom.xml : properties

`<properties>` : Permet la définition d'un ensemble de properties et de variables accessible dans le pom.xml

Exemple :

```
<properties>
  <project.build.sourceEncoding>utf-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>utf-8</project.reporting.outputEncoding>
</properties>
```

Les properties se déclarent avec la syntaxe `<ma.properties>Sa valeur<ma.properties>`

Le fichier pom.xml : properties

Les properties s'utilisent avec la syntaxe `${ma.properties}`

Exemple :

```
<version>${ma.version}</version>
```

Il existe un certain nombre de properties fournies par maven et qui peuvent être utilisées sans avoir à les définir dans la balise adéquate. En voici quelques unes :

- ${env.XXX} : Accès à la variable d'environnement XXX
- ${project.XXX} : Accès à un élément de configuration du projet, comme project.version, qui reprend la valeur de la version du projet
- ${settings.XXX} : Accès à une valeur définie dans le settings.xml de l'utilisateur

Le fichier pom.xml : properties

Certaines properties sont utilisées par certains plugins par défaut de maven. C'est par exemple le cas de :

```
<project.build.sourceEncoding>utf-8</project.build.sourceEncoding>
```

Qui définit l'encodage des source du projet, ici en UTF-8

Il faut donc les modifier avec parcimonie, et vérifier leur usage

Le fichier pom.xml : builds

La balise `<build>` sert principalement à la déclaration des plugins qui seront utilisés lors le compilation du projet et de la création de son livrable.

Elle contient elle même une balise `<plugins>`, elle même contenant une liste de balises `<plugin>`.

Chacune de ces dernières servira à la description du comportement d'un plugin.

Le fichier pom.xml : builds

Chaque balise `<plugin>` va posséder une partie commune, la balise `<artifactId>`, ainsi qu'une balise `<configuration>` qui lui est propre.

Ici l'exemple du **maven-compiler-plugin**, identifié par la balise `<artifactId>`, qui permet de spécifier en quelle version de Java sont écrites les sources, ainsi que la version des binaires de sortie.

```
<plugin>
   <artifactId>maven-compiler-plugin</artifactId>
   <configuration>
     <source>1.6</source>
     <target>1.6</target>
   </configuration>
 </plugin>
```

Le fichier pom.xml : profiles

Maven facilite fortement la portabilité d'un environnement (poste de développeur) à un autre (serveur d'intégration continue).
Toutefois, il arrive que pour rendre cette portabilité fonctionnelle partout, des ajustements soient nécessaires dans l'un ou l'autre des cas.

Les **profils** ont été inventés pour mettre ces différences de configuration ou d'exécution entre deux utilisations d'un même fichier pom.xml

Le fichier pom.xml : profiles

Ils se spécifient en paramètre de la ligne de commande avec l'option -p, séparés par des virgules :

mvn clean -P profile1,profile2

Ils peuvent également être activés automatiquement selon certaines conditions :

- Une activation par défaut choisie dans le settings.xml au moyen des balises `<activeProfiles>` et `<activeProfile>`
- La présence d'une balise `<activation>` ainsi que les conditions nécessaires dans la définition du profil dans le pom.xml

Le fichier pom.xml : profiles

Leur définition se fait au moyen de la balise `<profiles>`, elle même constituée d'une liste de `<profile>`. Les identifiants de chacun des profil ne doivent pas contenir de caractères spéciaux ou d'espacement

Exemple :

```
<profiles>
     <profile>
          <id>IdentifiantDeMonProfil</id>
          <!-- spécifité du profile -->
     <profile>
</profiles>
```

Le fichier pom.xml : profiles

Les profils peuvent surcharger les `<properties>`, `<plugins>` ou bien encore les `<resources>`. La liste complète est disponible dans la document de l'outil.

Il est impossible d'ajouter des dépendances spécifiques pour un profil.

Ils peuvent être définis directement dans le settings.xml de l'utilisateur, voire dans la configuration globale de l'outil, contenue dans ses binaires.
C'est toutefois une pratique déconseillée compte tenu de la non visibilité des modifications potentielles au sein des sources du projet.

Le fichier pom.xml : dépendances

Les dépendances peuvent être gérées au sein de la balise `<dependencies>`, contenant elle même une liste de `<dependency>`.

```
<dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
          …
    </dependency>
<dependencies>
```

Le fichier pom.xml : dépendances

Comme pour notre projet, une dépendance est identifiée par son `<groupId>` et son `<artifactId>`.

Il est également nécessaire de préciser la `<version>` nécessaire, cette dernière pouvant avoir un impact direct sur les éléments que contient la librairie.

Pour qu'une dépendance soit téléchargée, et effectivement intégrée au projet, il est nécessaire d'atteindre **au moins la phase compile du lifecycle default.**

Le fichier pom.xml : dépendances

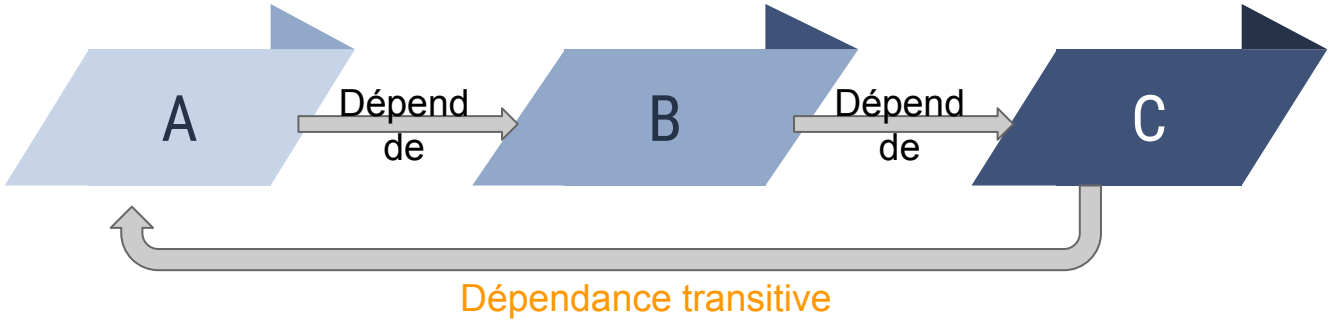
Un `<scope>` peut être ajouté à une dépendance. Il conditionne l'usage de cette dernière, ainsi que sa présence dans le packaging.

Valeurs possibles :

- **Compile** : Dépendance nécessaire pour le projet et packagée dans l'application. Transmise aux projets fils (cf. héritage)
- **Provided** : Dépendance nécessaire mais fournie au classpath par un autre élément, comme le serveur web
- **Test** : Dépendance servant uniquement dans les tests de l'application et pas à son exécution
- **Runtime** : Dépendance requise uniquement pour l'exécution, pas la compilation
- **System** : Similaire à provided, mais nécessite que la source soit spécifiée
- **Import** : Très spécifique, utilisable uniquement pour l'héritage de projets

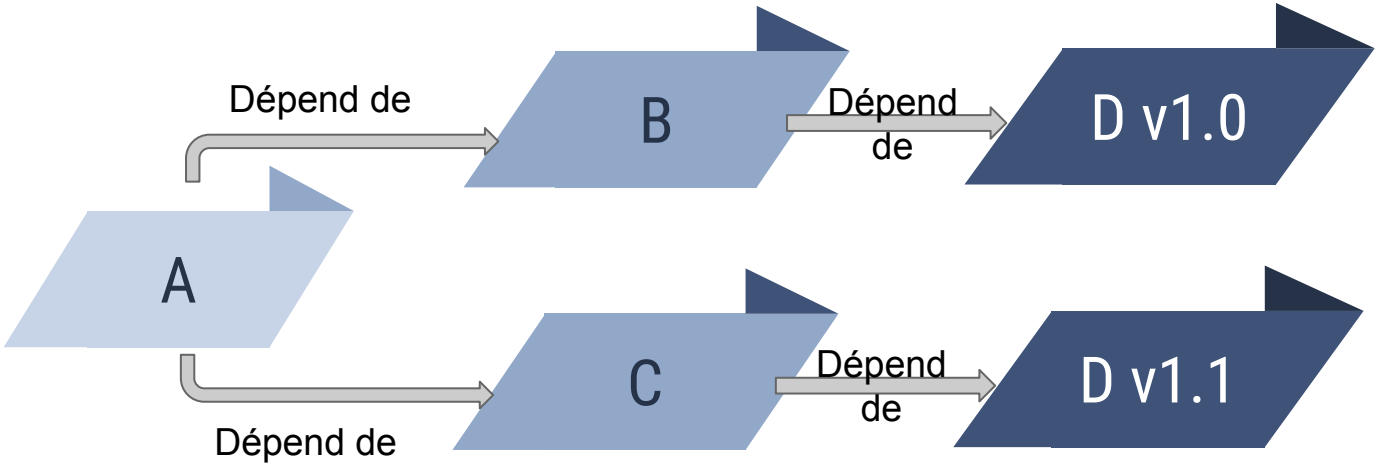
Le fichier pom.xml : transitivité des dépendances

Une dépendance A d'une application B possède elle même des dépendances (C par exemple) . Ces dernières sont amenées dans l'application B par transitivité des dépendances.



Le fichier pom.xml : transitivité des dépendances

La transitivité des dépendances **favorise le dependency Hell** !



Si les classes de D sont incompatibles, laquelle utiliser ?

Le fichier pom.xml : exclusions

Lors de l'import de dépendances dans le pom, il est possible d'exclure des dépendances amenées par transitivité:

```
<dependency>
    <groupId>groupe.test</groupId>
    <artifactId>dependanceA</artifactId>
    <version>1.0</version>
    <exclusions>
        <exclusion>
            <groupId>groupe.test2</groupId>
            <artifactId>dependanceB</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Le fichier pom.xml : exclusions

`<exclusions>` contient une liste de balises `<exclusion>`. Chacune d'elles doit contenir le `<groupId>` et le `<artifactId>` de la dépendance que l'on souhaite exclure.

Il est également possible d'utiliser le wildcard "*****" pour exclure toutes les dépendances d'un groupe.

NB: pour connaître l'état de la hiérarchie des dépendances, en incluant les dépendances transitives, il est possible d'utiliser la commande **mvn dependency:tree**.

La majorité des IDE intègre une version plus esthétique du résultat de cette commande.

Le fichier pom.xml : héritage

Comme dans les langages objets et donc Java, il existe une notion d'héritage entre les différents projets : **les pom parents**

La déclaration d'héritage se fait dans le pom fils, au moyen de la balise `<parent>`.

```
<parent>
   <groupId>groupe.test</groupId>
   <artifactId>projetParent</artifactId>
   <version>1.0</version>
</parent>
```

Comme pour les dépendances, les projets parents sont à identifier avec leur `<groupId>`, leur `<artifactId>` et leur `<version>`

Le fichier pom.xml : héritage

Les projets fils héritent des properties, dépendances et plugins de leurs parents

Les properties peuvent être écrasées de la même manière qu'en les définissant normalement.

Les dépendances amenées par le père peuvent être managées avec la balise `<dependencyManagement>`.

Les plugins amenés par le père peuvent être managées avec la balise `<pluginManagement>`.

Le fichier pom.xml : dependencyManagement

Ne sont affectées que les dépendances présentes dans le pom parent

```
<dependencyManagement>
   <dependencies>
     <dependency>
       <groupId>groupe.test</groupId>
       <artifactId>dependanceA</artifactId>
       <version>1.1</version>
     </dependency>
   </dependencies>
</dependencyManagement>
```

Cette déclaration vient surcharger la version de la dépendance A du pom parent en version 1.1.

Le mécanisme pour `<pluginManagement>` est le même.

Spring

Qu’est ce que c’est?

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

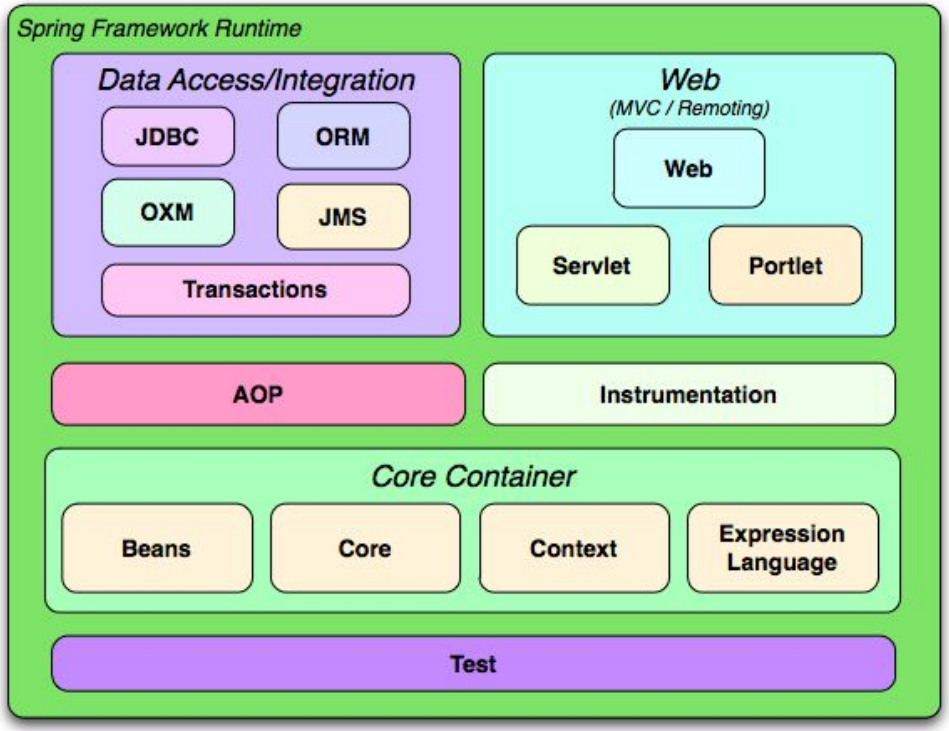
“Il y a une solution spring pour à peu près tout”

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.

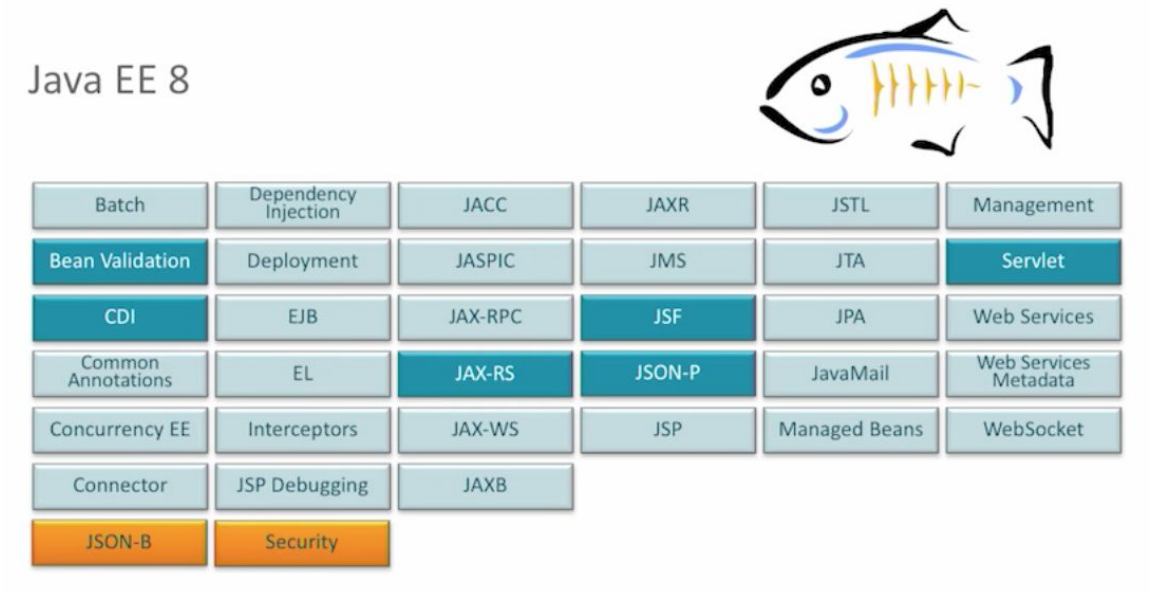
Une découpe en modules

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)

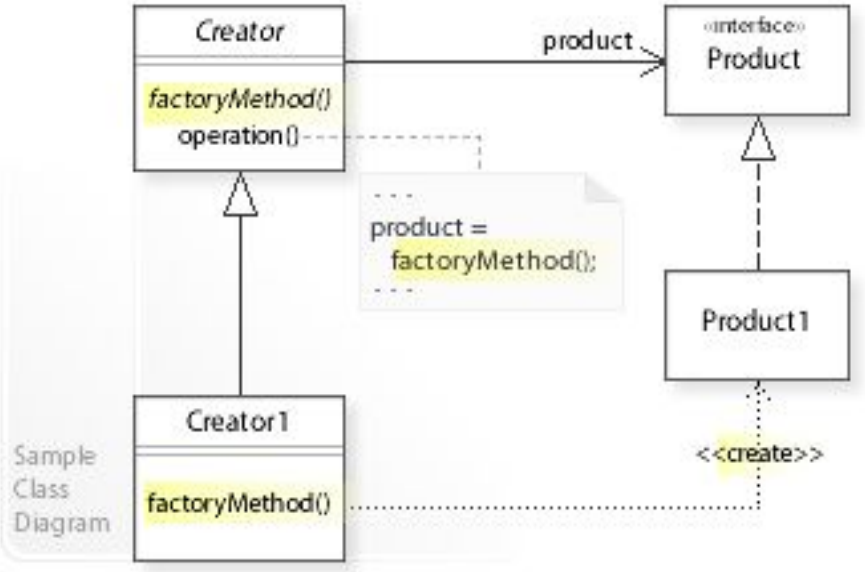
La classe concrète est inconnue de l'appelant :

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

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

On parle de **Contrat de Service**.

La factory


Creator
factoryMethod()
operation()
product
«interface»
Product
product =
factoryMethod();
Product1
Creator1
factoryMethod()
<<create>>
Sample
Class
Diagram

La factory concrètement

```
public class ListeFactory{


   public static List<String> creerListe(int type){
       switch(type){
           case 1 : return new ArrayList<String>();
           case 2 : return new LinkedList<String>()
           default: throw new IllegalArgumentException();
       }
    }
}
```

La factory concrètement

```
List<String> maListe  = ListeFactory.creerListe(1);
```

Pour faire simple, cela revient à déléguer à une classe le rôle d'instanciation d'une autre, le plus généralement abstraite.

La classe concrète derrière n'est ainsi jamais connue des appelants.

La factory concrètement

Spring = factory

En tant que conteneur léger, Spring fait fonction de factory.

Il va instancier des objets, le plus souvent implémentant des interfaces, et les fournir à d'autres classes appelantes.

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

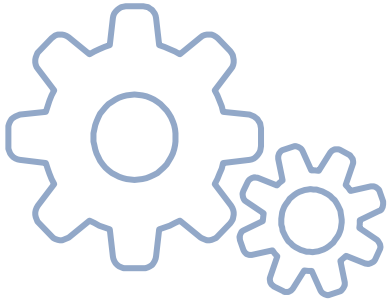
Via annotation sur la classe:

```
@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

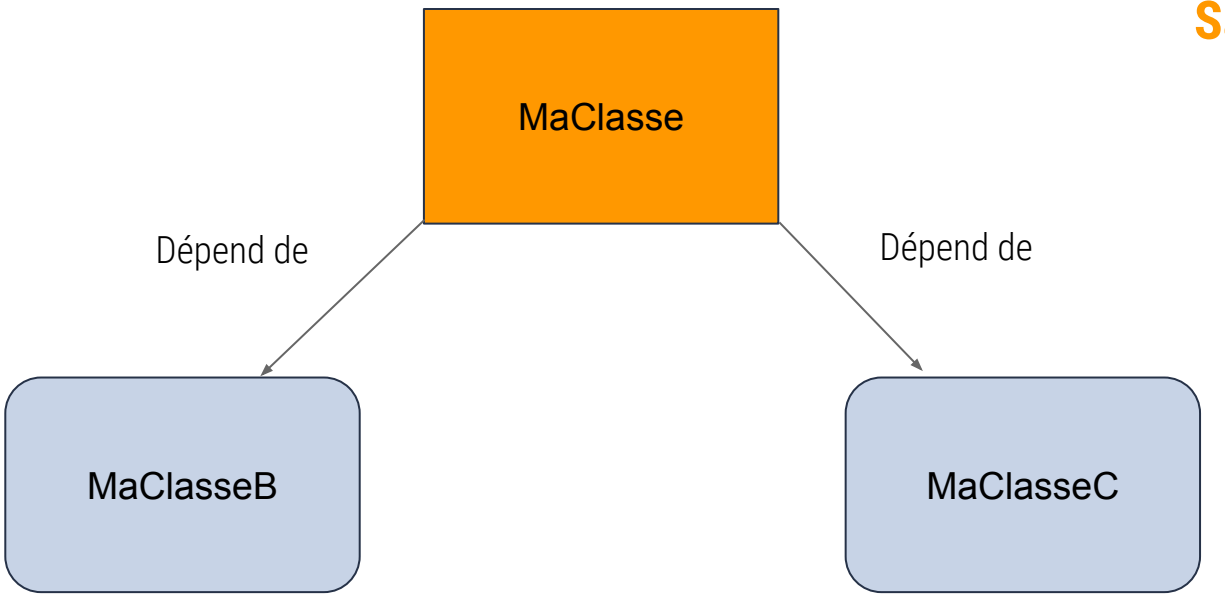
Soit MaClasse, MaClasseB et MaClasseC trois classes reliées comme ci dessous :

```
public class MaClasse {

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

Inversion de contrôle (IOC)

Sans IOC



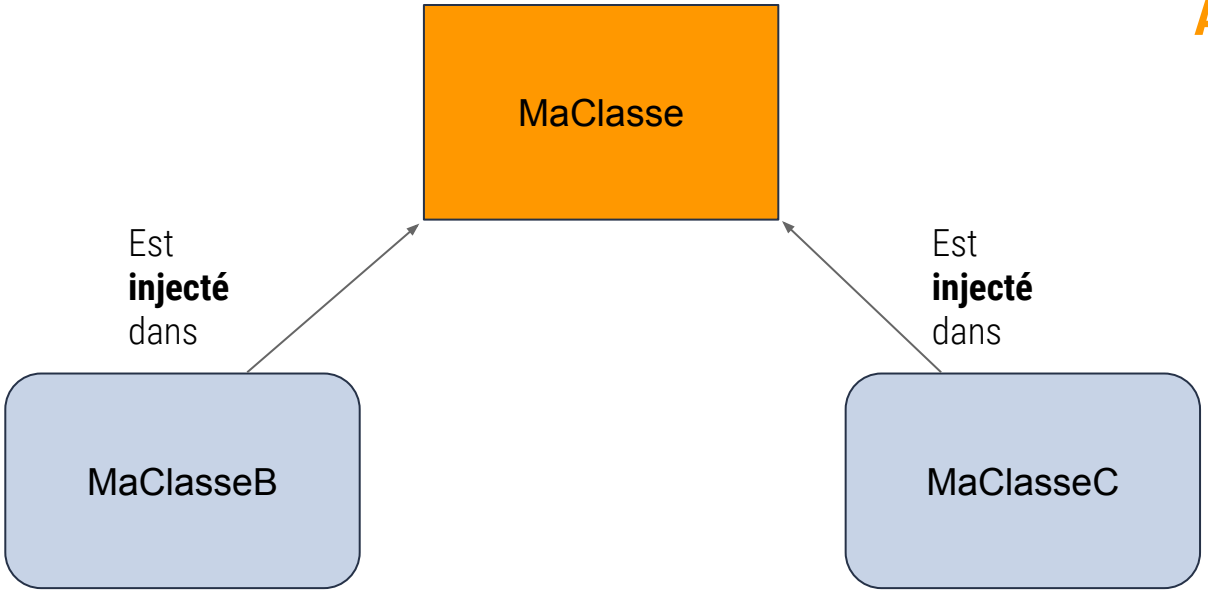
Inversion de contrôle (IOC)

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

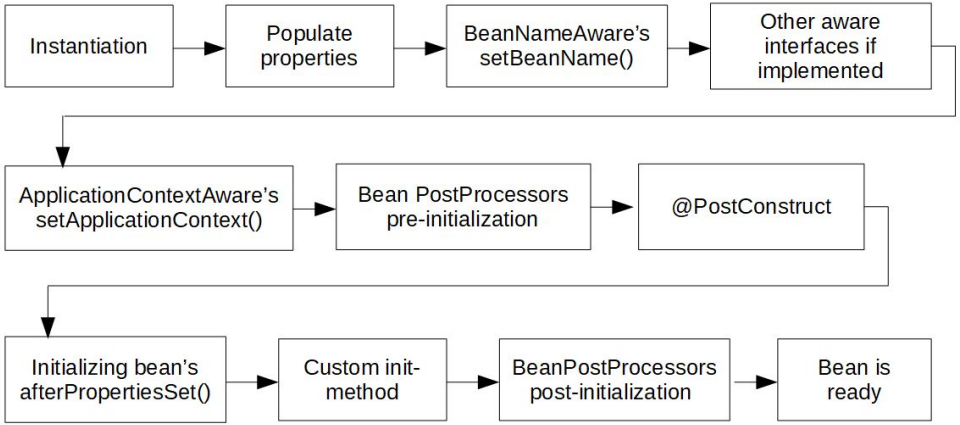
“A matter of Life and Death”

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

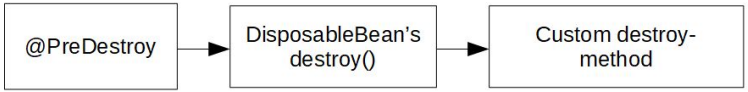
Il s’agit d’un enchaînement d’étapes composée d’appels de méthodes, internes à Spring ou pas, qui vont permettre :

- 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

Injection de dépendances

La déclaration d'une dépendance à injecter se fait manuellement comme ci dessous, en XML:

```
<bean name="testBean" class="com.test.Bean">
  <property name="testProperty" value="Hello world"/>
</bean>
```

Ici un bean de type "Bean" sera créé et ajouté au contexte Spring. Sa propriété "testProperty" sera valorisée par Spring avec le chaine de caractère "Hello world".

Injection de dépendances

L'équivalent en configuration Java se ferait ainsi, dans une classe de Configuration:

```
@Bean
public Bean testBean() {
    Bean bean = new Bean();
    bean.setTestProperty("Hello World");
    return bean;
}
```

Cette déclaration remplace uniquement la phase d'instanciation du cycle de vie d'un Bean.

Chaînage des injections

Un bean peut en utiliser un autre

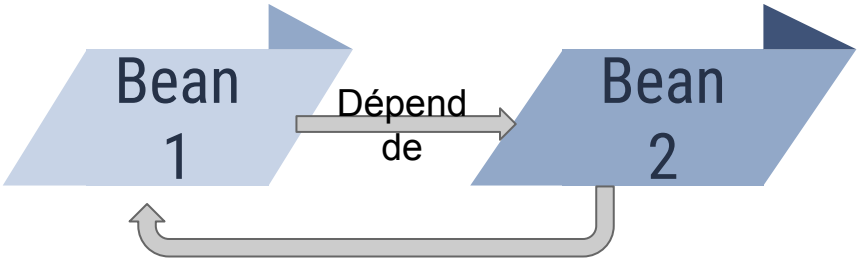
On parle alors de chaînage des injections, ou de chaînage de beans

Chaînage des injections

Cycle d'injections

Tout cycle dans les injections est à proscrire car insoluble pour Spring.

En effet, un bean n'est injectable que s'il est pleinement configuré.



Insoluble

Pour être instancié, Bean1 a besoin de Bean2, qui a besoin de Bean1, etc

Chaînage des injections manuel

En XML le chaînage manuel des injections se fait avec l'attribut "ref", référençant un bean par son "id".

```
<bean id="test" name="testBean" class="com.test.Bean">
  <property name="testProperty" ref="bean2"/>
</bean>
<bean id="bean2" class="com.test.Bean2">
</bean>
```

Dans l'exemple précédent, le bean "testBean" se voit injecter un bean de type Bean2, dont l'identifiant est "bean2" dans la property "testProperty"

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

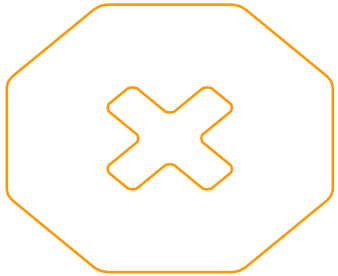
L'annotation **@Autowired** ouvre la voie de l'injection dite "automatique".

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.

“L’icône de Java c’est du café, parce que tu peux t’en servir un le temps que ca démarre”

Blague de développeur

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.

Design Pattern Singleton

Il ne peut y en avoir qu'un

Ce design pattern garantit qu'il n'existe qu'une seule et unique instance d'une classe appelée "Singleton"

Design Pattern Singleton concrètement

```
public class Singleton {
    private Singleton() {}
    private static Singleton INSTANCE = null;
    public static Singleton getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Singleton();
        }
        return INSTANCE;
    }
}
```

Design Pattern Singleton concrètement

Usage

L'usage du singleton permet, entre autre, de garantir l'accès concurrent et unique à une ressource partagée.

Exemple: un fichier système, un tableau en mémoire, etc.

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

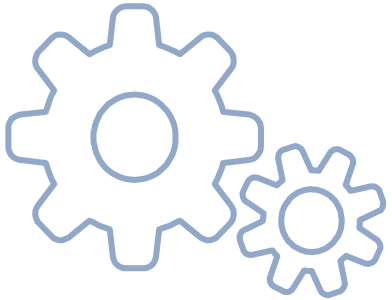
Mais pas top!

“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

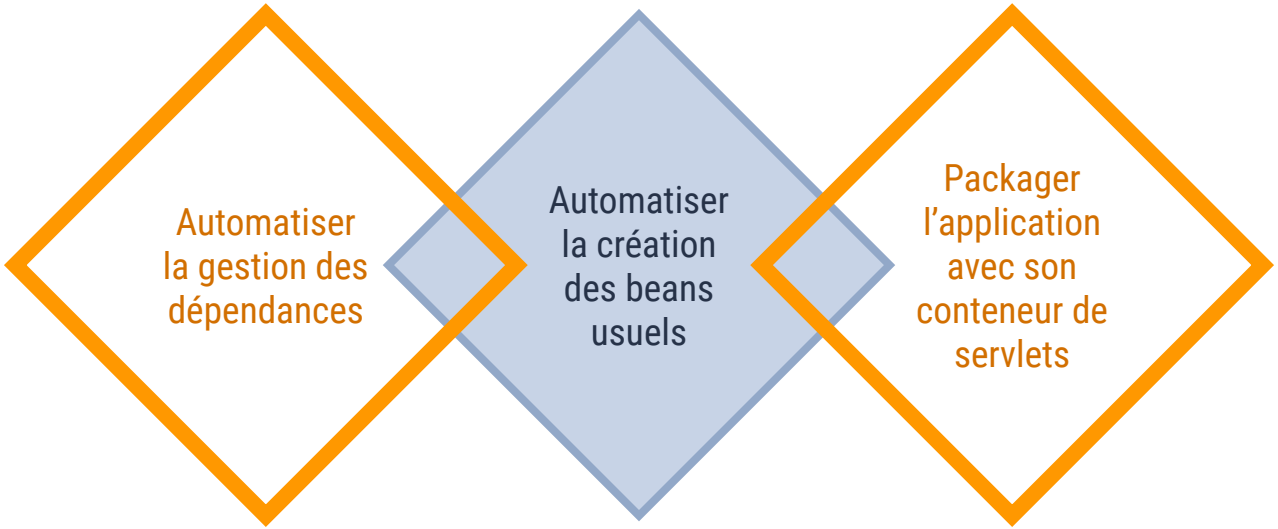
Automatiser ce qui ne produit pas de valeur

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
```

Auto Configuration

Ces deux fichier de configuration sont équivalents

Ils permettent de créer une Datasource (source de données) vers une base de données embarquée en mémoire H2, dont l'identifiant de connexion est "test".

Les clés de paramétrage sont les mêmes, seule la syntaxe et la structure du fichier diffèrent.

Le choix est à l'appréciation du développeur !

Auto Configuration

RTFM!

La liste des properties est en constante évolution, au grés des versions :

https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

Auto Configuration

Autres mécanismes

L'utilisation du fichier de properties n'est pas le seul mécanisme permettant l'auto configuration : Il existe d'autres déclencheurs:

- La présence ou l'absence d'une classe dans le classpath
- La présence ou l'absence d'un bean dans le contexte Spring
- La présence ou l'absence d'une ressource
- Le fait d'être en contexte Web, ou pas
- Plus généralement du SpEL

Créer son auto configuration

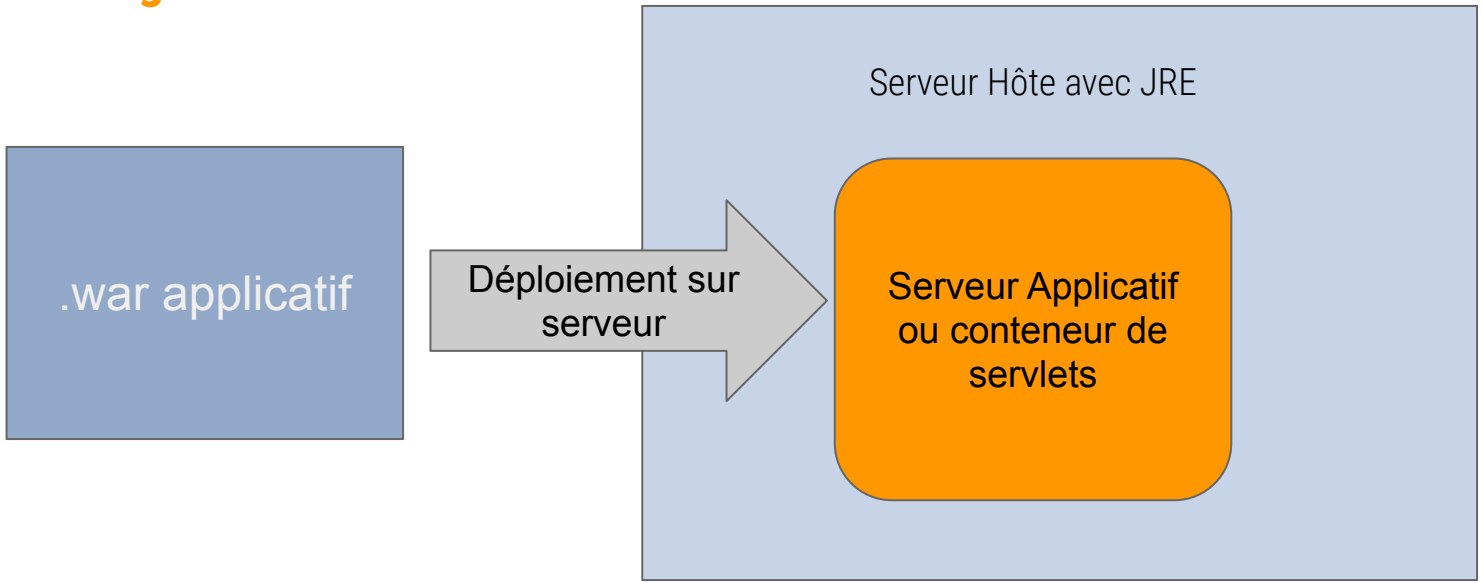
Il est possible de créer sa propre auto configuration.

Cette nécessité apparaît surtout lors du développement d'une librairie commune, plus que dans le cas d'une application à proprement parler.

L'objectif reste le même: **produire de la valeur**

Packaging Spring Boot

Cas général : le .war



Packaging Spring Boot

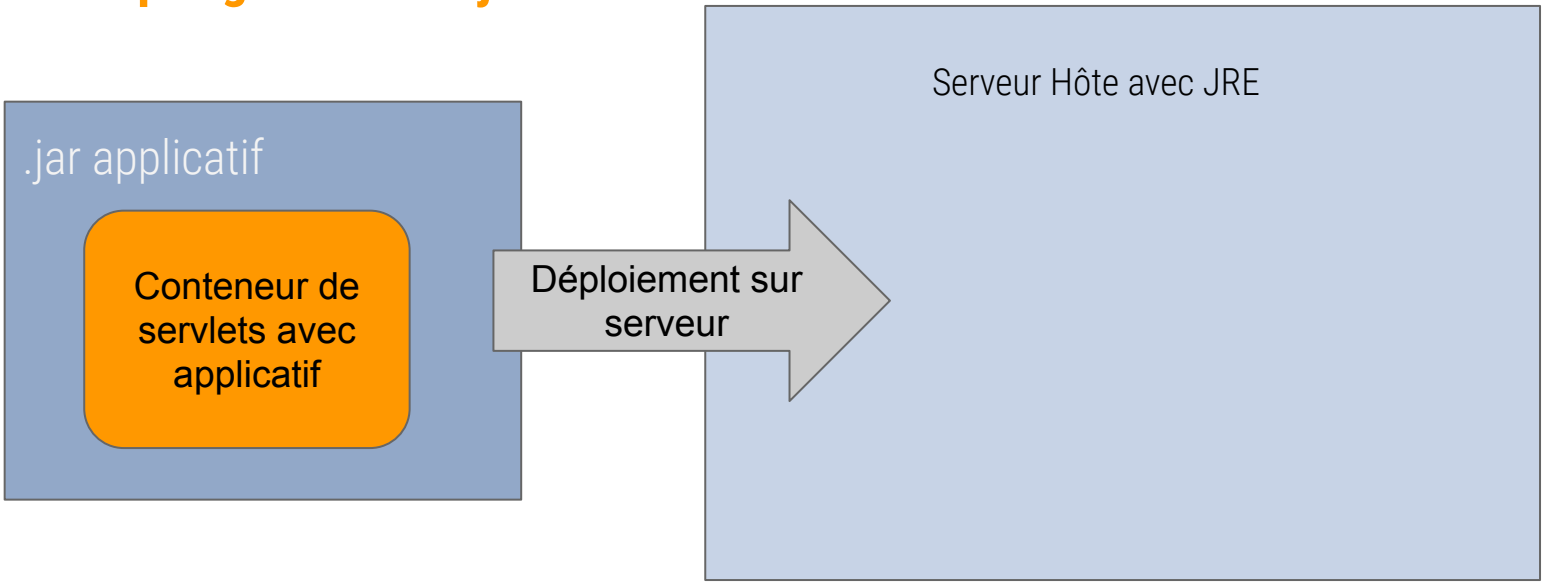
Usuellement, une application JEE ou Spring se package sour la forme d'un .war ou d'un ear.

Celui ci est déployé dans une conteneur de servlets ou un serveur d'applications sur une machine hôte, équipée d'une JRE.

Le process de livraison consiste à refournir le livrable au serveur, qui se charge du déploiement.

Packaging Spring Boot

Spring Boot : un .jar



Packaging Spring Boot

Par défaut et en contexte WEB, Spring Boot va packager les applications sous la forme d'un .jar contenant :

- L' applicatif en lui même, sous la forme d'un .war
- Un conteneur de servlets, par défaut Tomcat.

Le livrable embarque donc le serveur de déploiement.

Le process de livraison consiste à refournir le .jar complet, sur une machine équipée d'une simple JRE.

Packaging Spring Boot

Avantages

Le livrable est presque suffisant. Il n'est pas nécessaire de prévoir des étapes d'installation fastidieuses.

La compatibilité entre l'applicatif et le serveur de déploiement est assurée : si l'application se lance sur un poste de développement, tout laisse à penser que l'environnement de production sera opérationnel.

Packaging Spring Boot

Avantages

L'utilisation d'un applicatif par serveur facilite le cloisonnement des applications entre elles :

La chute ou la surcharge d'une application sur un serveur peut entraîner ce dernier dans sa chute. Ici, une seule application sera impactée.

Cette particularité facilite la scalabilité horizontale et l'orientation dite **"Micro Services"**

Packaging Spring Boot

Inconvénients

La taille du livrable est le principal défaut:

Dépendances comprises, un .war applicatif oscille généralement entre 30 et 50 Mo.

Le .jar Spring Boot est au minimum à 80Mo. Ceci s'explique par la gestion des super dépendances embarquées : beaucoup des librairies incluses dans le livrable ne sont pas nécessaires pour l'applicatif.

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.

Des classes de Configuration

D’autres classes dites de Configuration peuvent venir épauler la Main Class pour la mise en place des éléments, beans, etc.

Elles sont identifiées par Spring via une annotation class-level:

```
@Configuration
```

Elles peuvent par exemple définir de nouveaux beans hors auto configuration, modifier des paramétrages par défaut, etc.