

Introduction à Spring Boot

Spring se présente comme un éco système Java: Spring propose de nombreux outils et bibliothèques pour développer des applications serveurs, des web services... Son développement est organisé autour de plusieurs projets, voir le site **spring.io**. Spring Boot est l'un de ces projets, on peut le considérer comme un outil majeur pour le développement de web services et de **micro services**.

STS fonctionne avec le JDK version 1.8 et plus.

Le développement Spring peut se faire soit en installant :

- le plugin **Spring Tools for Eclipse IDE** en passant par le **Marketplace** d'une version d'Eclipse déjà installée:
Menu Help> Eclipse Marketplace, rechercher spring tools, installer le plugin Spring Tools...
- le logiciel **Spring Tool Suite** (STS). La suite STS se présente comme une version d'Eclipse avec les plugins Spring Boot installés.
Télécharger le fichier `spring-tool-suite-....` et exécuter le (par un double clic).
L'exécutable **SpringToolSuiteN.exe** est situé dans le dossier décompressé obtenu (N = N° de version).
Exécuter le fichier **SpringToolSuiteN.exe** et vérifier dans le menu **Windows|Preferences|Java|Installed JREs** l'utilisation/l'installation d'une JRE ou d'un JDK. Sinon installer le JDK.

Spring Boot propose une "façon de faire" pour créer des projets Spring et notamment des micro services. Cela se traduit par la création automatique d'un fichier Maven (ou Gradle) pour assurer la configuration du projet intégrant les diverses bibliothèques nécessaires aux fonctionnalités choisies par le développeur et ainsi gérer les dépendances requises pour assurer ces fonctionnalités.

La gestion des dépendances d'un projet est effectuée à l'aide d'un mécanisme appelé **Spring Boot Starter** qui permet à partir d'un formulaire de sélectionner les fonctionnalités utilisées dans le projet. Ce choix étant fait, le **Starter** générera automatiquement le fichier des dépendances nécessaires au fonctionnement du projet (fichier pom.xml pour Maven).

De plus, Spring Boot propose un mécanisme d'auto configuration. Par exemple, si le projet utilise une base de données, Spring Boot est capable de générer le Data Source correspondant.

Il y a 2 façons "équivalentes" pour créer un projet Spring Boot:

- soit utiliser en local **STS**, solution utilisée dans le cours,
- soit utiliser **Spring initializr** en allant sur la page <https://start.spring.io/>

Présentation du mécanisme IOC

Un projet SpringBoot s'exécute dans un environnement spécial dédié appelé **containeur spring**. Le **containeur** fournit un environnement d'exécution. Un containeur propose des fonctionnalités que le programmeur peut utiliser pour simplifier sa programmation. Un containeur est responsable du cycle de vie de l'application et de fournir les fonctionnalités demandées par cette application.

Les fonctionnalités proposées par le containeur et utilisées par l'application sont appelées les dépendances.

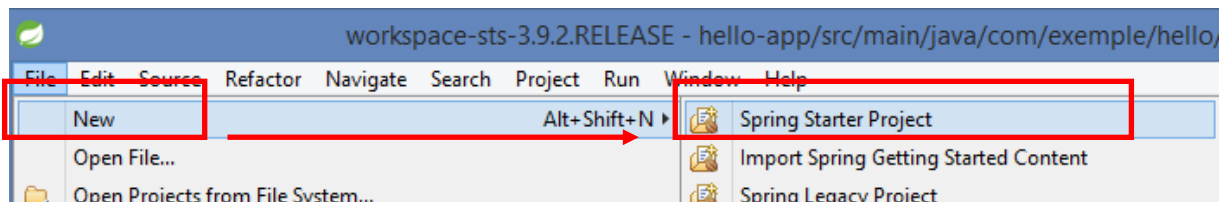
Par exemple, un objet pour fonctionner nécessite un autre objet, le conteneur est alors responsable d'instancier cet autre objet et de "l'injecter" dans celui qui l'utilise.

Dans un projet classique, le programme est responsable de l'instanciation des objets utiles au fonctionnement, généralement avec l'opérateur **new** mais aussi quelque fois en utilisant une factory dédiée. Chaque instanciation est contrôlée par le programme java et correspond à une instruction.

SpringBoot propose le mécanisme **Inversion Of Control IOC** qui délègue la création de certains objets au conteneur Spring. Le mécanisme IOC est mis en place en utilisant des annotations spécifiques. L'IOT est également appelé CDI: Context and Dependency Injection.

Création d'un projet minimal Spring Boot avec STS

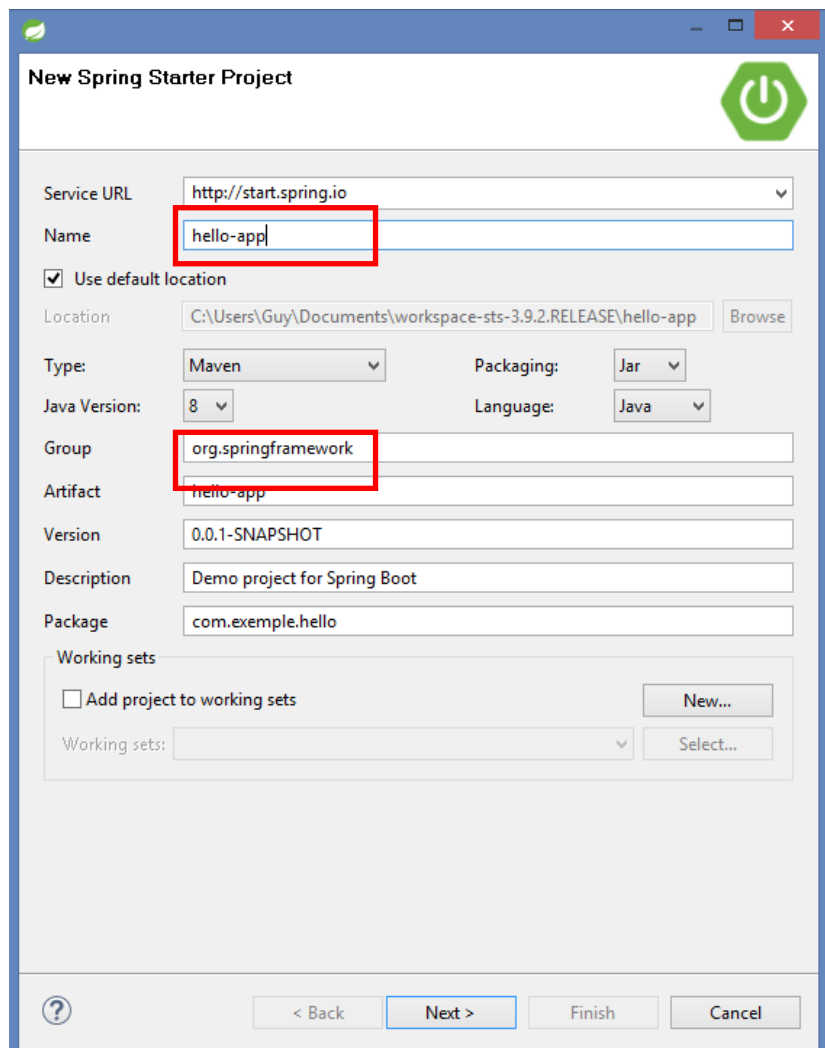
File > New > Spring Starter Project



Le nom du projet est donné dans le champ **Name** et se retrouve dans le champ **Artifact**.

Puis **Next**.

Le formulaire suivant permet de choisir les fonctionnalités du projet à réaliser.



New Spring Starter Project

Service URL:

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

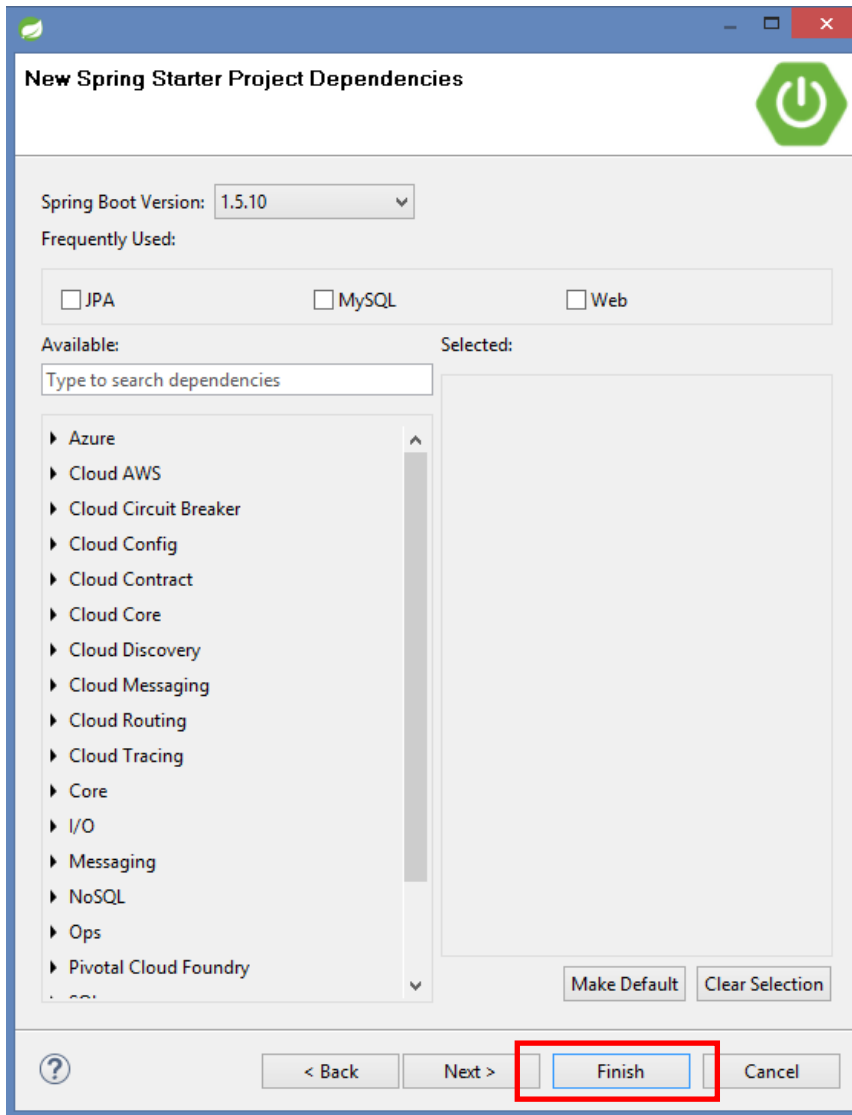
Description:

Package:

Working sets

☐ Add project to working sets

Working sets:



► On ne sélectionne rien puis **Finish**.

Le projet **hello-app** est créé. Il faut développer le projet dans la vue **Package Explorer**.
La classe **HelloAppApplication** a été créée.

```
package com.exemple.hello;

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

@SpringBootApplication
public class HelloAppApplication {

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

L'annotation **@SpringBootApplication** est une annotation composée qui regroupe les 3 fonctionnalités des 3 annotations suivantes :

- **@SpringBootConfiguration**: indique que cette classe fournit une configuration Spring Boot.

- **@ComponentScan**: indique à Spring Boot qu'il faut scanner le package courant et ses sous packages afin de trouver les **beans** annotés.
- **@EnableAutoConfiguration**: autorise Spring Boot d'analyser et de configurer les beans « auto configurés ».

Un **bean** est une recette pour créer un objet, instance d'une classe Java, géré par le conteneur Spring Boot.

Durant le démarrage, Spring Boot scanne le package courant et de ses sous packages pour détecter toutes les classes annotées, et instancier les objets pour les injecter dans les attributs ou les constructeurs d'autres beans Spring.

@Component est l'annotation de base pour marquer une classe comme composant (bean) SpringBoot.

@Service est utilisée pour marquer une classe métier ou une classe de service.

@Controller est utilisée pour marquer une classe qui contient des annotations @RequestMapping.

@RestController est utilisée pour marquer une classe contrôleur REST.

@Repository est utilisée pour marquée une classe DAO.

Par défaut un **bean** à une portée **singleton**. Cela veut dire que le **bean** est unique, le conteneur en crée une seule instance, il est partagé par les divers objets qui l'utilisent, le conteneur gère le cycle de vie d'un bean à **portée** singleton.

Une application simple en ligne de commande

On ajoute **implements CommandLineRunner** à la classe principale main du projet. Il faut définir la méthode **run()** de l'interface.

```
package com.exemple.hello;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloAppApplication implements CommandLineRunner{

    public static void main(String[] args) {
        SpringApplication.run(HelloAppApplication.class, args);
        System.out.println("Fin du main");
    }

    @Override
    public void run(String... args) throws Exception {
        // TODO Auto-generated method stub
        System.out.println("Début du run");
        Thread.sleep(5000); //attente de 5s
        System.out.println("Fin du run");
    }
}
```

Exécution

```
2021-05-13 ---
Début du run
```

```
Fin du run  
Fin du main
```

La méthode **main()** est essentielle pour exécuter cette application en tant qu'application SpringBoot. L'annotation **@SpringBootApplication** est suffisamment explicite. Le code de l'application en ligne de commande doit être développé dans la méthode **run()** définie.

Ajout d'un bean

Ajout d'un 1^{ère} bean exécuté dès que l'application est chargée.

```
package com.exemple.hello;  
  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.stereotype.Component;  
  
@Component  
public class HelloCommandLineRunner implements CommandLineRunner {  
  
    @Override  
    public void run(String... args) throws Exception {  
        // TODO Auto-generated method stub  
        System.out.println("Exécution de CommandLineRunner");  
    }  
}
```

► Exécuter l'application: **cliquez droit sur le projet | Run As | Spring Boot App**

@ComponentsScan: Spring Boot scanne les packages (package courant + enfants) pour trouver les composants, par exemple ici le package `com.exemple` et ses sous packages.

La classe **HelloCommandLineRunner** annotée **@Component** (située dans l'arborescence) est détectée. Cette classe implémente l'interface **CommandLineRunner**, Spring Boot exécute automatiquement la méthode **run()**.

La figure suivante donne toutes les annotations utilisées pour marquer des classes qui seront détectées par Spring Boot.

```
@Component      @Bean  
  
@Service        @Configuration  
  
@Controller     @Repository  
  
@RestController
```

Ajout d'un 2^{ème} bean exécuté dès que l'application est chargée.

```
package com.exemple.hello;  
  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.stereotype.Component;
```

```

@Component
public class HelloCommandLineRunnerBis implements CommandLineRunner{

    @Override
    public void run(String... args) throws Exception {
        // TODO Auto-generated method stub
        System.out.println("Exécution de CommandLineRunner 2");
    }
}

```

► Exécuter l'application. Observer les résultats affichés.

► Pour modifier l'ordre, il faut que les 2 classes implémentent l'interface **Ordered**.

```

package com.exemple.hello;

```

```

import org.springframework.boot.CommandLineRunner;
import org.springframework.core.Ordered;
import org.springframework.stereotype.Component;

```

```

@Component
public class CommandLineRunner implements CommandLineRunner, Ordered {

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Exécution de CommandLineRunner");
    }

    @Override
    public int getOrder() {
        return 2;
    }
}

```

Faire la même modification pour CommandLineRunner1 avec:

```

@Override
public int getOrder() {
    return 1;
}

```

► Exécuter l'application. Observer les résultats affichés.

Ajout d'un bean implémentant un service

► L'interface: le service consiste à afficher un message

```

package com.exemple.hello;

```

```

public interface HelloService {
    public void sayHello(String name) ;
}

```

► Classe qui implémente le service

```

package com.exemple.hello;

```

```

public class ConsoleHelloService implements HelloService {
    private String prefix ;
    private String suffix ;

    public ConsoleHelloService(String prefix, String suffix) {
        super();
        this.prefix = (prefix!=null? prefix : "Bonjour");
        this.suffix = (suffix!=null? suffix : "!");
    }
    public ConsoleHelloService() {
        prefix = "Bonjour";
        suffix = "!";
    }
    @Override
    public void sayHello(String name) {
        String msg = String.format("%s %s%s", prefix, name, suffix);
        System.out.println(msg);
    }
}

```

Modification du HelloCommandLineRunner

```

package com.exemple.hello;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class HelloCommandLineRunner implements CommandLineRunner{

    private HelloService helloService ;
    @Override
    public void run(String... arg0) throws Exception {
        // TODO Auto-generated method stub
        helloService.sayHello("Runner");
    }
    public HelloCommandLineRunner(HelloService helloService) {
        this.helloService = helloService;
    }
}

```

Exécuter l'application: l'exécution de l'application *plante* car la classe **HelloCommandLineRunner** a besoin d'un objet **HelloService** qui n'est pas ici automatiquement créé et injecté dans le constructeur de **HelloCommandLineRunner**.

L'annotation @Bean

Cette annotation est utilisée pour annoter une **méthode** qui retourne un objet (le bean). Cet objet Bean est créé au démarrage de l'application, il est unique, le conteneur crée une seule instance de cette classe dans l'application: c'est un **singleton**, sa visibilité (portée, scope) est au niveau de l'application. Il sera créé puis injecté dans les classes qui l'utilisent, comme il est unique toutes ces classes partageront le même objet.

L'annotation **@Bean** doit être détectée, pour cela elle doit être utilisée dans une classe annotée **@Configuration**, **@SpringBootApplication** ou **@Component**.

Le bean est ici injecté dans le paramètre **helloService** du constructeur de la classe **HelloCommandLineRunner** **HelloCommandLineRunner(HelloService helloService)**.

```
package com.exemple.hello;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class HelloAppApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloAppApplication.class, args);
    }
    @Bean
    public HelloService genereHelloService() {
        return new ConsoleHelloService("Hello", "!");
    }
}
```

► Exécuter le projet et justifier les résultats.

Un Bean est un Singleton (par défaut)

L'annotation **@Bean** est portée sur une méthode qui retourne un objet.

Cet objet est un Bean: un singleton, objet unique dans l'application, créé par Spring Boot au démarrage de l'application. Le bean est injecté dans le constructeur de la classe **Component** **HelloCommandLineRunner**.

On modifie le code des classes précédentes afin de montrer le fonctionnement du singleton et notamment la "persistance" de ses attributs, ce qui prouve que ce bean est unique.

On ajoute un attribut de type entier dans la classe **ConsoleHelloService**. On incrémente cet entier à chaque utilisation du bean et affiche cet entier dans la méthode **sayHello()**.

```
public class ConsoleHelloService implements HelloService {
    private String prefix ;
    private String suffix ;
    private int N =0; //L'entier
    public ConsoleHelloService(String prefix, String suffix) {
        super();
        this.prefix = (prefix!=null? prefix : "Bonjour");
        this.suffix = (suffix!=null? suffix : "!");
        N++;
        System.out.println(N);
    }
    public ConsoleHelloService() {
        prefix = "Bonjour";
        suffix = "!";
        N++;
        System.out.println(N);
    }
}
```



```

@Override
public void sayHello(String name) {
    N++;
    String msg = String.format("%s %s %d", prefix, name, suffix,N);
    System.out.println(msg);
}
}

```

```

@Component
public class HelloCommandLineRunner implements CommandLineRunner{

    private HelloService helloService ;
    @Override
    public void run(String... arg0) throws Exception {
        // TODO Auto-generated method stub
        helloService.sayHello("Runner");
        helloService.sayHello("Runner");
    }
    public HelloCommandLineRunner(HelloService helloService) {
        this.helloService = helloService;
    }
}

```

```

@Component
public class HelloCommandLineRunner1 implements CommandLineRunner{

    private HelloService helloService ;
    @Override
    public void run(String... arg0) throws Exception {
        // TODO Auto-generated method stub
        helloService.sayHello("Runner one");
        helloService.sayHello("Runner one");
        helloService.sayHello("Runner one");
        helloService.sayHello("Runner one");
    }
    public HelloCommandLineRunner1(HelloService helloService) {
        this.helloService = helloService;
    }
}

```

Affichage après exécution de l'application

```

2021-05-11 16:43-----
1
2021-05-11 16:43-----
Hello Runner! 2
Hello Runner! 3
Hello Runner one! 4
Hello Runner one! 5
Hello Runner one! 6
Hello Runner one! 7

```

► Exécuter le projet et expliquer les résultats.

Cet exemple montre bien le fonctionnement du Bean `HelloService`: il est unique dans l'application, créé au démarrage de l'application, injecté dans les Component `HelloCommandLineRunner`. La valeur de N est incrémentée à chaque appel de la méthode `sayHello()` et maintenu entre 2 appels successifs de cette méthode.

Autre façon d'injecter un bean: l'annotation @Autowired

► On met en commentaire ou supprime l'annotation **@Bean** et la méthode associée pour créer le bean dans la classe HelloAppApplication.

```
@SpringBootApplication
public class HelloAppApplication {
    public static void main(String[] args) {
        SpringApplication.run(HelloAppApplication.class, args);
    }
}
/* @Bean
public HelloService genereHelloService() {
    return new ConsoleHelloService("Hello", "!");
}
*/ }
```

► On porte l'annotation **@Component** sur la classe **ConsoleHelloService**.

```
@Component
public class ConsoleHelloService implements HelloService {
    private String prefix ;
    private String suffix ;
    ---
}
```

► Dans chaque classe **HelloCommandLineRunner** on porte l'annotation **@Autowired** sur l'attribut **HelloService**. L'injection se fera sur l'attribut annoté **@Autowired**.

```
package greta78.cda;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class HelloCommandLineRunner implements CommandLineRunner{

    @Autowired
    private HelloService helloService ;
    @Override
    public void run(String... arg0) throws Exception {
        // TODO Auto-generated method stub
        helloService.sayHello("Runner");
        helloService.sayHello("Runner");
    }
}
/* public HelloCommandLineRunner(HelloService helloService) {
    this.helloService = helloService;
} */
}
```

Exécuter le programme.

La portée prototype **@Scope("prototype")** d'un bean

Ajouter l'annotation **@Scope("prototype")** devant la classe **ConsoleHelloService**

```
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("prototype")
```

```
@Scope("prototype")
public class ConsoleHelloService implements HelloService {
    private String prefix ;
    private String suffix ;
    ---
}
```

Exécuter l'application.

Résultats affichés:

```
1
1
2021-05-13 10:20:11.---
Bonjour Runner! 2
Bonjour Runner! 3
Bonjour Runner one! 2
Bonjour Runner one! 3
Bonjour Runner one! 4
Bonjour Runner one! 5
```

On constate maintenant que 2 beans ont été créés: la portée "**prototype**" entraîne l'instanciation d'un nouveau bean et son injection chaque fois qu'un client demande ce bean.

Spring Data supporte 6 portées (scope): **singleton**, **prototype** et les 4 suivantes **request**, **session**, **application** et **websocket** qui doivent être utilisées dans le contexte web.

Autre exemple pour montrer l'injection

Créer un nouveau projet. La classe avec la méthode main :

```
package com.example.demo;

import java.util.Arrays;

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

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        var ctx = SpringApplication.run(DemoApplication.class, args);
        System.out.println("Nombre de beans: "+ctx.getBeanDefinitionCount());
        var names = ctx.getBeanDefinitionNames();
        Arrays.sort(names);
        for (String n:names)
            System.out.println(n);
    }
}
```

Le mot réservé **var** permet lors de l'écriture du code de ne pas se soucier du type de la variable ctx, la JVM induit son type à la compilation.

Ici, on affiche le nombre de beans créés par le conteneur SpringBoot, puis on récupère la liste des noms de tous ces beans et on l'affiche.

Exécuter l'application et observer les résultats.

Ajouter au projet une classe **MaClasseA** annotée avec **@Component**.

```
package com.example.demo;

import org.springframework.stereotype.Component;

@Component
```

```
public class MaClasseA {
    public void affiche() {
        System.out.println("Je suis le bean A");
    }
}
```

Exécuter l'application. Justifier le nombre de beans par-rapport au cas précédent.

L'instanciation du bean `maClasseA` est mise en évidence en utilisant l'annotation `@PostConstruct` sur une méthode qui sera appelée dès que l'objet sera instancié.

```
@Component
public class MaClasseA {
    @PostConstruct
    public void affiche() {
        System.out.println("Je suis le bean A");
    }
}
```

Exécuter l'application. Observer le message supplémentaire affiché.

On montre maintenant l'injection du bean `maClasseA` dans un nouveau composant.

Ajouter au projet une classe `UsingBeanA` annotée avec `@Component`.

```
package com.example.demo;

import javax.annotation.PostConstruct;
import org.springframework.stereotype.Component;

@Component
public class UsingBeanA {
    private MaClasseA maClasseA;
    public UsingBeanA(MaClasseA maClasseA) {
        this.maClasseA=maClasseA;
    }
    @PostConstruct
    public void afficheUsing() {
        System.out.println("Je suis usingBeanA");
        maClasseA.affiche();
    }
}
```

L'injection est ici faite dans le constructeur.

Exécuter l'application. Observer les messages affichés.

Bean stateless ou stateful, composant métier

On a vu qu'un bean **singleton** est partagé par tous les objets "client" qui ont demandé son instanciation et qu'il conserve ses attributs et donc son état. C'est un problème si le client doit utiliser les attributs (l'état) du singleton, car cet état est partagé et manipulé par tous les objets client qui l'ont référencé. C'est pour cette raison que dans la pratique, ses attributs –son état– ne sont pas utilisés. Spring conseille donc d'utiliser les singletons sans leurs attributs, uniquement leurs méthodes et donc exploiter ces singletons comme des beans **"stateless"**.

En revanche, chaque objet client qui le demande dispose de son propre bean **prototype**, les attributs du bean prototype ont donc des valeurs cohérentes pour l'objet client qui l'utilise. Spring conseille donc d'utiliser des prototypes pour créer des beans **"stateful"**, c'est à dire que l'état du bean est maintenu et cohérent entre 2 appels successifs de ces méthodes.

Les beans servent notamment à implémenter les composants "métier" d'une application. Un composant métier qui ne contient que des méthodes peut être défini dans un bean singleton, en utilisant ce bean comme un composant stateless, c'est-à-dire sans utiliser ces attributs qui n'ont pas de cohérence car ils sont partagés par d'autres objets de l'application.

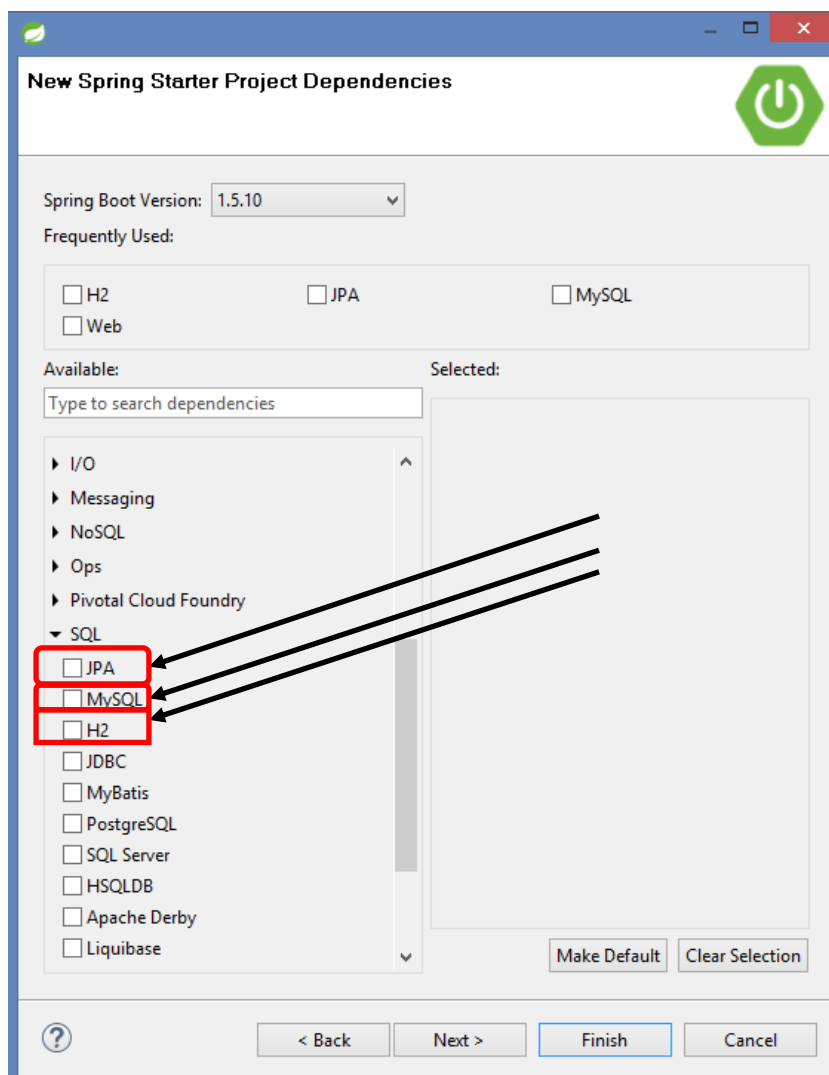
Les beans prototype –stateful– pourront être utilisés pour implémenter des composants métier avec des méthodes métier mais aussi avec des propriétés utiles.


Projet avec la base de données embarquée H2

H2 est une base de données "en mémoire". Elle est recrée à chaque (re)démarrage de l'application.

- Créer un nouveau projet comme précédemment.
- Sélectionner l'item **SQL** puis **JPA** et **H2** dans la liste déroulante.

Le formulaire présente les derniers choix effectués (Frequently Used:).



 **Version 2.6.2**
Spring Data JPA
H2 Database

► Créer une classe entité Customer

```
package com.exemple.hello;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;
    private String firstName;
    private String lastName;

    protected Customer() {}
    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String toString() {
        return String.format(
            "Customer[id=%d, firstName='%s', lastName='%s']", id, firstName, lastName);
    }
    public Long getId() {
        return id;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
}
```

► Créer une interface CustomerRepository

```
package com.exemple.hello;

import java.util.List;
import org.springframework.data.repository.CrudRepository;

public interface CustomerRepository extends CrudRepository<Customer, Long> {
    List<Customer> findByLastName(String lastName);
    List<Customer> findByFirstName( String surname);
}
```

Le projet **Spring Data** répond aux besoins des accès aux bases de données. Spring Data contient 3 interfaces que l'utilisateur peut étendre pour accéder aux données. Ces interfaces sont les DAO de la base manipulée.

On présente ici l'interface **CrudDirectory<T,ID>** : **CrudDirectory** est une interface générique, le 1^{er} paramètre T correspond au type de l'entité manipulée, le 2^{ème} paramètre correspond au type de la clef primaire de la table correspondante.

L'interface **CrudDirectory<T,ID>** permet l'auto génération des méthodes CRUD de base:

| | | |
|--------------------------------|----------------------|---|
| long | | count() Returns the number of entities available. |
| void | | delete(T entity) Deletes a given entity. |
| void | | deleteAll() Deletes all entities managed by the repository. |
| void | | deleteAll(Iterable<? extends T> entities) Deletes the given entities. |
| void | | deleteById(ID id) Deletes the entity with the given id. |
| boolean | | existsById(ID id) Returns whether an entity with the given id exists. |
| Iterable<T> | | findAll() Returns all instances of the type. |
| Iterable<T> | | findAllById(Iterable<ID> ids) Returns all instances of the type with the given IDs. |
| Optional<T> | | findById(ID id) Retrieves an entity by its id. |
| <S S | extends T> | save(S entity) Saves a given entity. |
| <S Iterable<S> | extends T> | saveAll(Iterable<S> entities) Saves all given entities |

De plus, en respectant une certaine syntaxe, on peut déclarer de nouvelles méthodes dans une interface qui étend **CrudDirectory**, voir l'interface **CustomerRepository** ci-dessus. Spring Data recherche les noms des méthodes qui contiennent des mots clefs comme find...By..., read...By..., query...By..., count...By..., et get...By... puis analyse le reste de la déclaration de ces méthodes afin d'en générer automatiquement l'implémentation.

► Créer une classe **CustomerLineRunner** implements **CommandLineRunner**

```
package com.exemple.hello;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class CustomerCommandLineRunner implements CommandLineRunner {
```

```

@Autowired
private CustomerRepository base ;
@Override
public void run(String... arg0) throws Exception {
    // TODO Auto-generated method stub
    base.save(new Customer("Jack", "Bauer"));
    base.save(new Customer("Chloe", "O'Brian"));
    base.save(new Customer("Kim", "Bauer"));
    base.save(new Customer("David", "Palmer"));
    base.save(new Customer("Michelle", "Dessler"));
    System.out.println("Exécution de CommandLineRunner");
    List<Customer> liste = (List<Customer>) base.findAll();
    System.out.println("Nombre d'éléments:"+liste.size());
    for (Customer c : liste)
        System.out.println(c.toString());
    liste = base.findByLastName("Bauer");
    for (Customer c : liste)
        System.out.println("Bauer: "+c.toString());
}
}

```

- **@Component** ⇒ la classe est détectée au chargement de l'application,
- **..... implements** CommandLineRunner ⇒ la méthode run() est exécutée,
- **@Autowired**
 - private CustomerRepository base ;
 - ⇒ **Injection** du repository (dépôt) avec l'annotation **@Autowired**
- **base.save(...)** utilisation de la méthode save pour persister un Customer,

► Exécuter le projet et justifier les résultats.

Ajout d'une méthode pour supprimer plusieurs objets Customer

```

public interface CustomerRepository extends CrudRepository<Customer, Long> {
    List<Customer> findByLastName(String lastName);
    List<Customer> findByFirstName(String surname);

    @Transactional
    List<Customer> removeByLastName(String lastname);
}

```

Il faut annoter **@Transactional** la méthode sinon erreur.

Spring Data génèrera automatiquement l'implémentation de cette méthode.

Ajout d'une méthode en utilisant JPQL:

Il faut utiliser l'annotation **@Query**.

```

public interface CustomerRepository extends CrudRepository<Customer, Long> {
    .....
    @Query("select p from Customer p where p.lastName = ?1 and p.firstName = ?2")
    List<Customer> findByLastNameAndFirstName(String lastName, String fistName);
}

```

Spring Data génèrera automatiquement l'implémentation de cette méthode.

De nombreux mots clefs existent pour créer les noms des méthodes, voir le lien <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/> pour la documentation Spring Data. Rechercher dans cette documentation Spring Data le chapitre Query Creation.

Test de ces méthodes: ajouter les instructions suivantes

```
public void run(String... arg0) throws Exception {
    ----
    liste = base.findByLastNameAndFirstName("Palmer","David");
    System.out.println("\nNombre d'éléments:"+liste.size());
    for (Customer c : liste)
        System.out.println(c.toString());

    base.removeByLastName("Bauer");
    System.out.println("\nAprès suppression:");
    for (Customer c : (List<Customer>) base.findAll())
        System.out.println("Bauer: "+c.toString());
}
```

► Exécuter le projet et justifier les résultats.

Le fichier src/main/resources/data.sql

On peut peupler la base en plaçant dans **src/main/resources/data.sql** des instructions SQL

```
insert into customer values(1, 'Jack', 'Bauer');
insert into customer values(2,'Chloe', 'O'Brian');
insert into customer values(3,'Kim', 'Bauer');
insert into customer values(4,'David', 'Palmer');
insert into customer values(5,'Michelle', 'Dessler');
```

L'apostrophe (single quote) O'Brian est désactivée en ajoutant une 2^{ème} apostrophe devant.

► Supprimer les instructions inutiles de la classe **CustomerCommandLineRunner** et tester.

Injection "auto détectée"

On modifie la classe **CustomerCommandLineRunner** en supprimant **@Autowired** et en créant un **constructeur**.

```
@Component
public class CustomerCommandLineRunner implements CommandLineRunner {
    // @Autowired
    private CustomerRepository base ;
    @Override
    public void run(String... arg0) throws Exception {
        ----
    }
    public CustomerCommandLineRunner(CustomerRepository base) {
        this.base = base;
    }
}
```

► Exécuter le projet et vérifier les résultats.

Projet avec la base de données MySQL

🌟* Version 2.6.2

On reprend les mêmes classes que pour H2 mais en utilisant MySQL à la place.

Spring Data JPA

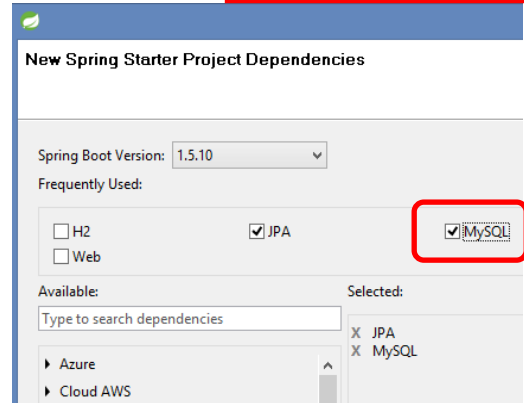
MySQL Driver

► Créer le projet avec les propriétés JPA et MySQL.

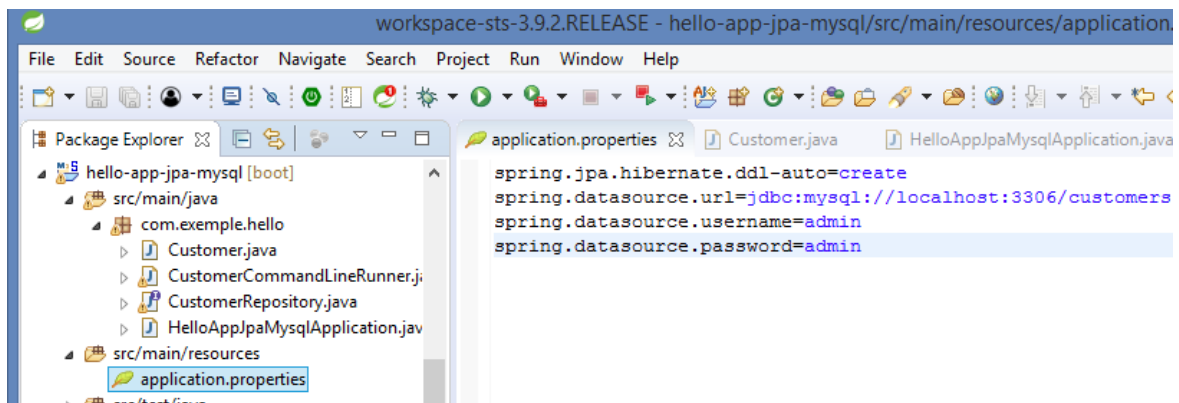
► Copier les fichiers utiles du projet "H2" dans ce nouveau projet.

► Créer une base de données **customers**.

► Remplir le fichier **src/main/resources/application.properties** avec les instructions suivantes:



```
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:mysql://localhost:3306/customers?serverTimezone=UTC
spring.datasource.username=admin
spring.datasource.password=admin
```



Le champ **spring.jpa.hibernate.ddl-auto** peut prendre les valeurs suivantes:

- **none** : c'est la valeur par défaut, aucun changement dans la structure de la base,
- **update** : Hibernate changes la structure de la base en accord avec les entités,
- **create** : crée la structure à chaque fois, mais ne la détruit pas à la fermeture,
- **create-drop** : crée la base à chaque fois, mais la détruit quand la SessionFactory ferme.

En général, on exécute une 1ère fois avec **create**, puis après avec **none** ou **update** quand on veut effectuer des changements sur la structure de la base.

► Exécuter le projet une 1^{ère} fois et vérifier les résultats.

► Après avoir créé la base, tester en suite avec **spring.jpa.hibernate.ddl-auto = none**.

Le fichier **application.properties** permet de modifier la configuration de Spring Boot et des dépendances du projet. Le lien suivant <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html> donne la liste complète des propriétés et leurs valeurs par défaut.

Exercice

On utilise la base lesinvites (ou une autre à votre choix)

Créer l'application qui teste les fonctionnalités que les DAO du projet doivent assurer.

Par exemple, pour lesinvites:

- Ajouter un invite.
- Associer une invitation à un (des invites).
- Afficher tous les invites.
- Afficher toutes les invitations.
- Afficher les invitations d'un invité.
- Afficher les invités ayant la même invitation.
- Supprimer un invite avec ses invitations.
- ...