



SPRING



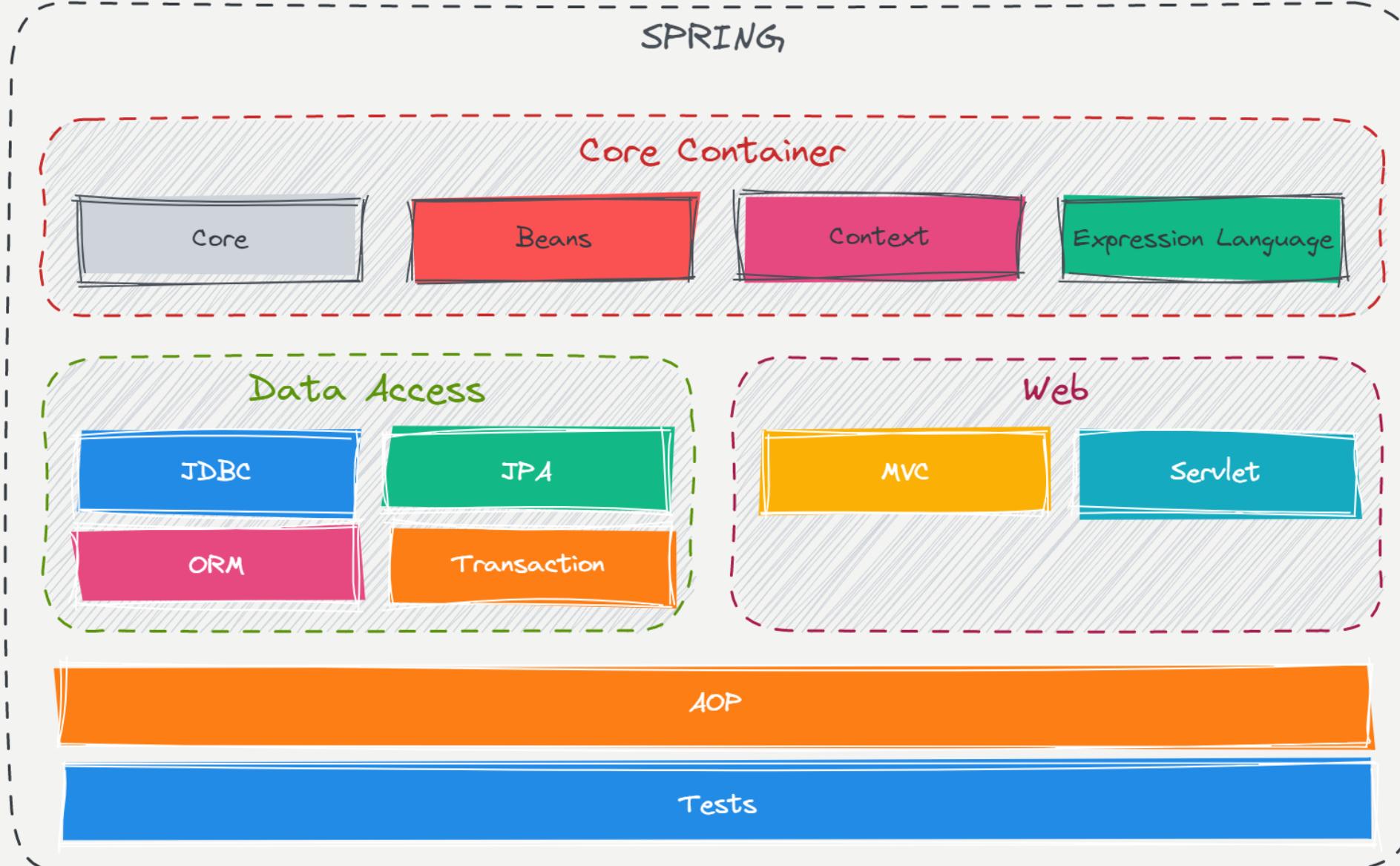
SPRING CORE

INTRODUCTION À SPRING

PRÉSENTATION DE SPRING

- Framework, standard industriel
- Constitué de modules
- Facilite le développement et les tests
- Spring gère des *JavaBean*, appelés *beans*
 - Classes **POJO** ou *JavaBean*
- Fournit les mécanismes
 - De fabrique d'objets (**BeanFactory**)
 - D'inversion de contrôle (**IoC – Inversion of Control**) : l'injection de dépendances

PRÉSENTATION DE SPRING



PRÉSENTATION DE SPRING

- Dépendance Maven
 - **spring-context**
- Maven chargera les dépendances
 - **spring-core** Cœur de **Spring**
 - **spring-beans** Conteneur **IoC** et fonctionnalités d'injection
 - **spring-context** Contexte de Spring (ApplicationContext)
 - **spring-expression** SpEL pour manipulation via un langage de requêtes **Spring**
 - **spring-aop** Aspect Oriented Programming

PRÉSENTATION DE SPRING

- **Dependency Injection (DI)**
- RAPPEL : On parle de dépendance entre objets lorsque
 - ClasseA a un attribut de type ClasseB
 - ClasseA est de type ClasseB, ou implémente InterfaceB
 - ClasseA dépend d'un type ClasseC qui lui-même dépend d'un type ClasseB
 - Une méthode de ClasseA appelle une méthode de ClasseB
- Les interfaces permettent de nous abstraire de l'implémentation finale
 - Mais nécessite toujours l'instanciation d'une classe concrète, l'instanciation de l'implémentation

PRÉSENTATION DE SPRING

- Soit les interfaces suivantes

```
public interface IMusicien {  
    public void jouer();  
}
```

```
public interface IInstrument {  
    public String son();  
}
```

PRÉSENTATION DE SPRING

- Soit les classes suivantes

```
public class Guitariste implements IMusicien {  
    private IInstrument instrument = new Guitare();  
  
    public void jouer() {  
        System.out.println("Le guitariste joue : " + this.instrument.son());  
    }  
}
```

```
public class Guitare implements IInstrument {  
    public String son() {  
        return "GLINK GLINK GLINK";  
    }  
}
```

C'est fonctionnel,
mais ce n'est pas bien parce que ce n'est pas son rôle d'instancier l'instrument !

PRÉSENTATION DE SPRING

- Dans un programme principal

```
public class Application {  
    public static void main(String[] args) {  
        IMusicien myMusicien = new Guitariste();  
        myMusicien.jouer();  
    }  
}
```

C'est fonctionnel,
mais ce n'est pas bien parce que ce n'est pas son rôle d'instancier le musicien !

PRÉSENTATION DE SPRING



- Le problème : chaque classe dépend d'une autre classe parce qu'elle doit l'instancier
 - Application dépend de Guitariste
 - Guitariste dépend de Guitare
- Si on ne veut plus d'un Guiistariste mais d'un Pianiste, il faudra changer les implémentations

PRÉSENTATION DE SPRING

- Pour résoudre ce problème, créons une Factory qui se chargera
 - D'instancier l'instrument

```
public class InstrumentFactory {  
    public static IInstrument getInstrument() {  
        return new Guitare();  
    }  
}
```

PRÉSENTATION DE SPRING

- Créons une Factory qui se chargera d'instancier le musicien

```
public class MusicienFactory {  
    public static IMusicien getMusicien() {  
        IMusicien musicien = new Guitariste();  
  
        musicien.setInstrument(InstrumentFactory.getInstrument());  
        return musicien;  
    }  
}
```

INJECTION DE DÉPENDANCES

- Remarque : on va donner au musicien sa dépendance à l'instrument

```
public class Guitariste implements IMusicien {  
    private IInstrument instrument;  
  
    public void setInstrument(IInstrument instrument) {  
        this.instrument = instrument  
    }  
}
```

PRÉSENTATION DE SPRING

- Dans notre application principale, nous avions ceci

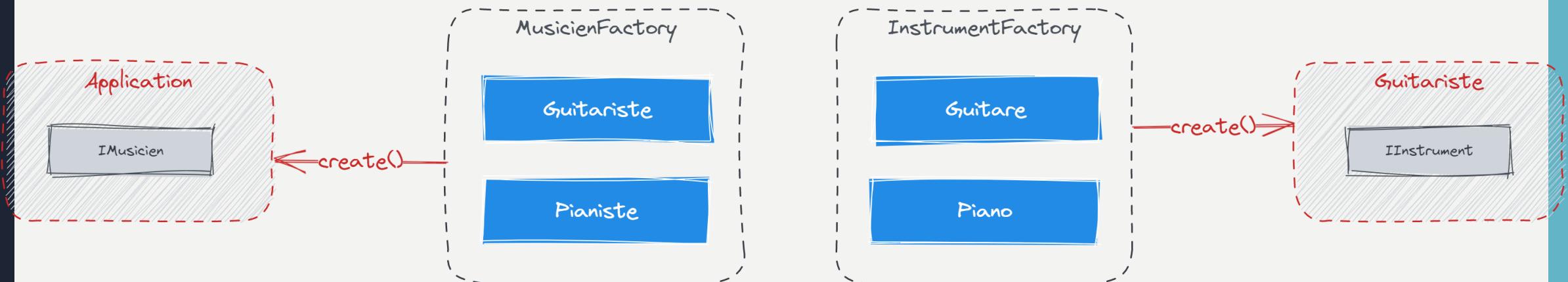
```
IMusicien myMusicien = new Guitariste();
```

- Remplacé par ça

```
IMusicien myMusicien = MusicienFactory.getMusicien();
```

La classe n'instancie plus, mais récupère sa dépendance via la **Factory** !

PRÉSENTATION DE SPRING



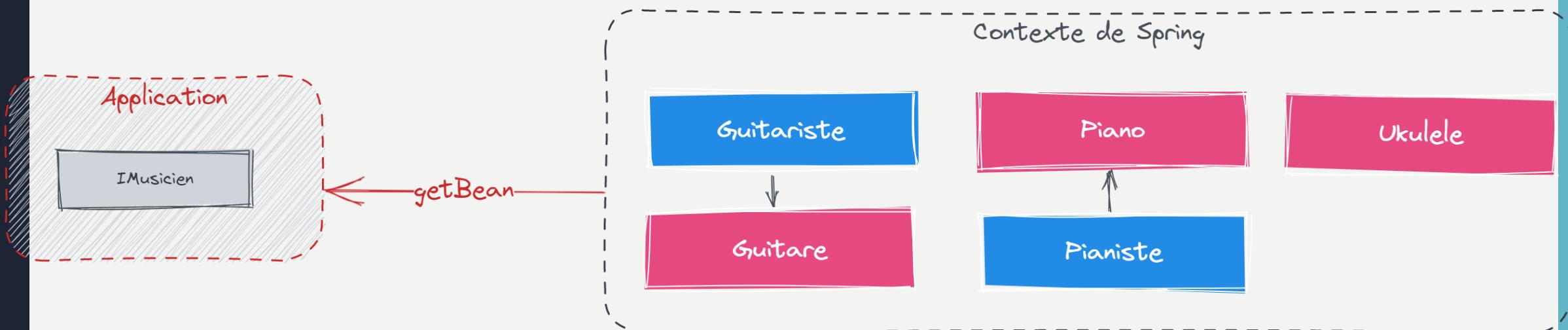
- Nos classes ont toujours des dépendances, mais elles sont récupérées depuis les Factories
 - Si l'implémentation est à changer, on ne modifie que les Factories !
 - Comme si on avait une configuration qu'on pouvait modifier !

PRÉSENTATION DE SPRING

- Si nous voulons changer les implémentations, seules les **Factories** sont à modifier
- Les classes **Musicien** et **Application** ne gèrent plus l'instanciation
 - C'est ce qu'on appelle l'injection de dépendances, le pattern **Inversion of Control (IoC)**
- C'est un travail fastidieux, en partie possible grâce à l'abstraction (interfaces)
 - C'est là que **SPRING** entre en jeu !

PRÉSENTATION DE SPRING

- Spring s'appuie sur le pattern **Inversion of Control**
- Permet de rendre indépendantes les couches techniques
 - **IoC** se charge d'instancier et de donner la référence créée !



PRÉSENTATION DE SPRING

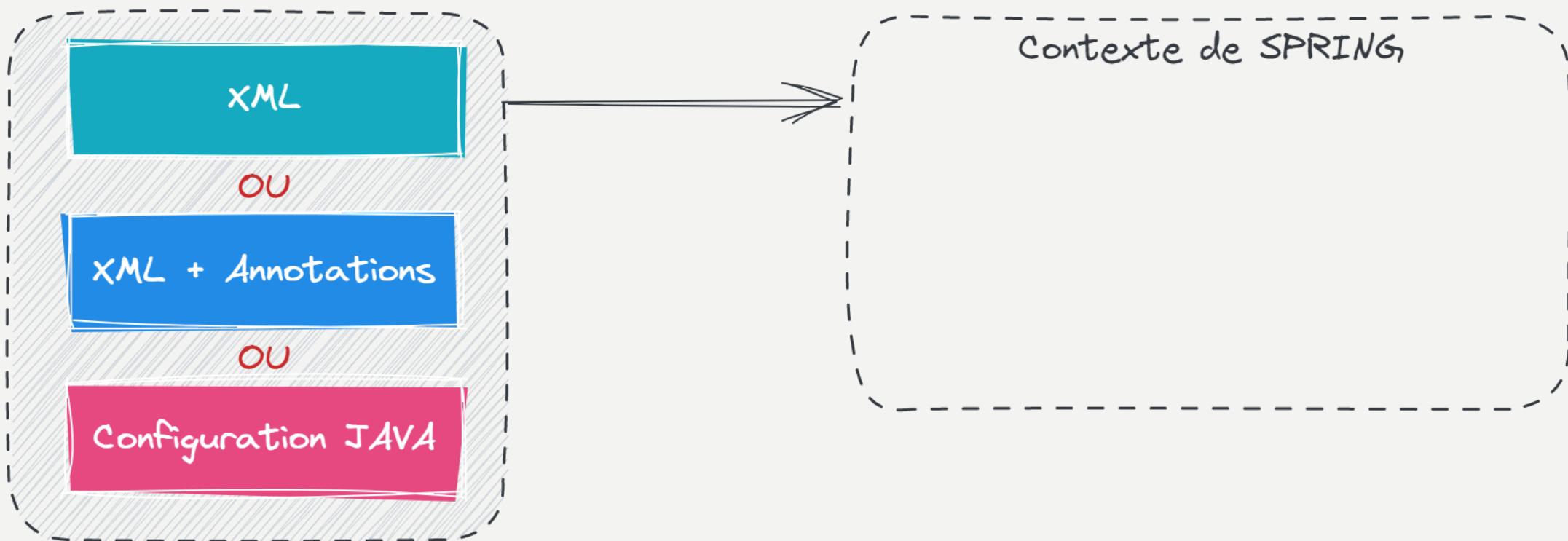
- L'injection de dépendances n'est possible que dans le contexte de **Spring Container**
 - Les objets y ont accès par un quelconque moyen
 - Les objets sont déjà dans le contexte de **Spring**
- **Spring** ne peut nous injecter que des objets qu'il manage
 - Depuis un programme principal (qui n'est pas géré par **Spring**)
 - Il faudra charger ce contexte pour récupérer les instances gérées par Spring
- **Spring** injectera les dépendances après l'instanciation des objets
 - Donc les références injectées ne sont pas disponibles dans le constructeur de l'objet

PRÉSENTATION DE SPRING

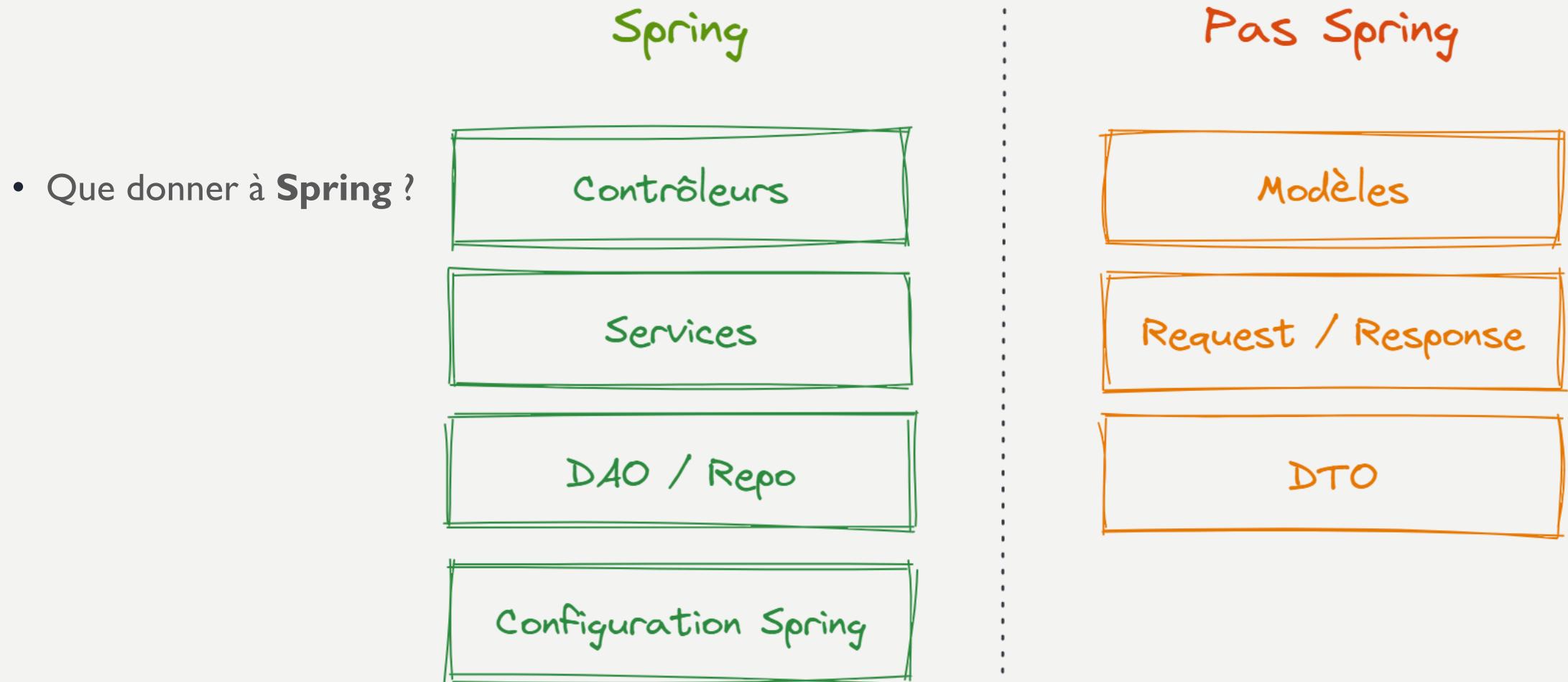
- Par chance, **Spring** ne va pas gérer toutes les instances de toutes les classes ...
- Il faut lui préciser
 - Quelle(s) instance(s) il doit manager
 - Quelle instance injecter dans quel attribut
- Ces précisions se font par déclaration (**XML** ou annotations **JAVA**)

PRÉSENTATION DE SPRING

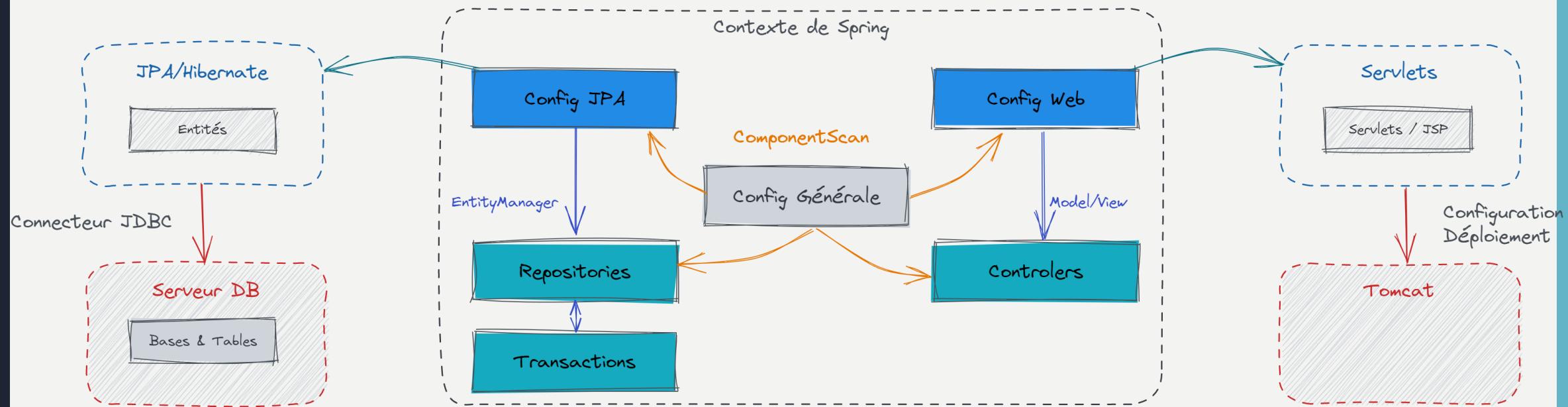
On configure SPRING



PRÉSENTATION DE SPRING



PRÉSENTATION DE SPRING





CONFIGURATION

CONFIGURATION DE SPRING

CONFIGURATION

- Pour configurer **Spring**, on utilise généralement un fichier par utilité
 - Configuration générale
 - Configuration Web
 - Configuration API
 - Configuration de la sécurité
 - ...

CONFIGURATION

- Pour créer une classe de configuration **SPRING**

```
@Configuration  
public class AppConfig {  
}
```

CONFIGURATION

- Pour déclarer un **bean**, on crée une méthode avec son type de retour
 - Le nom du **bean** (son identifiant) sera le nom de la méthode par défaut
 - On peut modifier ce comportement en donnant une valeur à l'annotation

```
@Configuration
public class AppConfig {
    @Bean
    public IInstrument guitare() {
        return new Guitare();
    }

    @Bean("guitariste2")
    public IMusicien guitariste() {
        return new Guitariste();
    }
}
```

CONFIGURATION

- Pour charger cette configuration dans l'application principale

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);  
IMusicien myMusicien = ctx.getBean(IMusicien.class);
```

- En chargeant par son nom uniquement

```
IMusicien myMusicien = (IMusicien)ctx.getBean("guitariste");
```

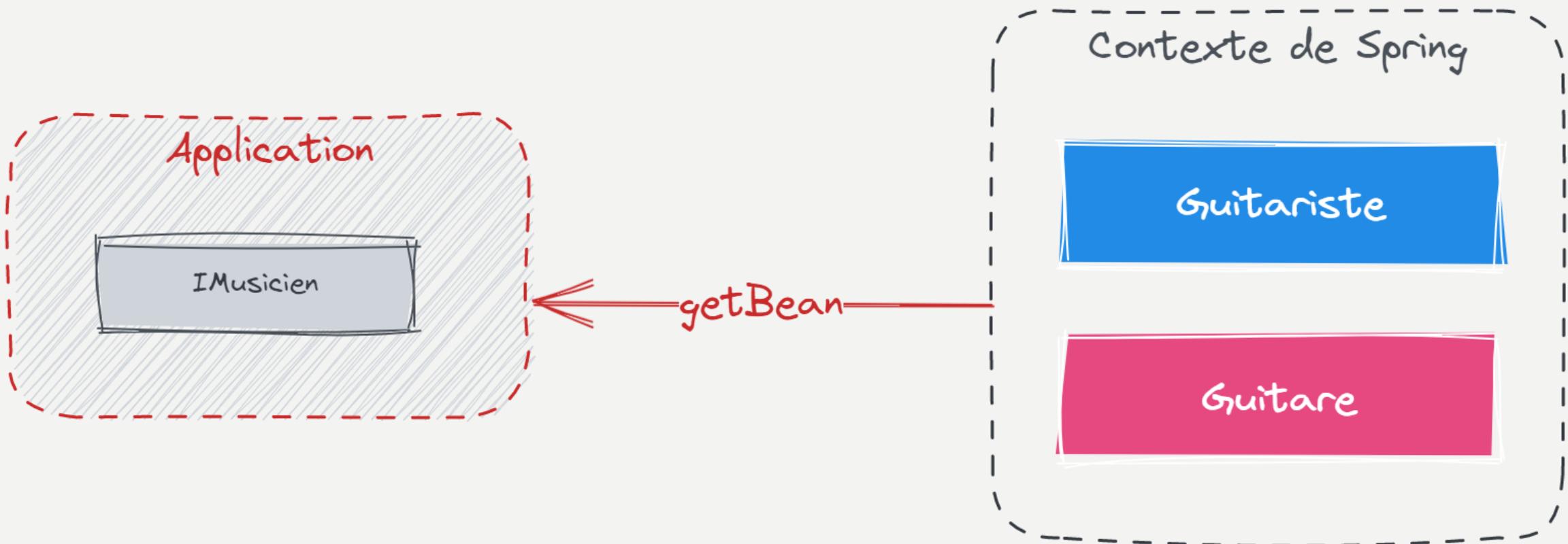
- En chargeant par son type uniquement

```
IMusicien myMusicien = ctx.getBean(IMusicien.class);
```

- En chargeant par son nom et son type

```
IMusicien myMusicien = ctx.getBean("guitariste", IMusicien.class);
```

CONFIGURATION



CONFIGURATION

- On peut inclure un fichier de configuration **XML** pour une configuration par classe
 - Annotation `@ImportResource`

```
@Configuration
@ImportResource("classpath:/application-context.xml")
public class AppConfig {
```

}

EXERCICE

- Créer un nouveau projet « formation-spring »
- Créer une classe principale `Application` avec sa méthode `main`
- Créer le modèle vu précédemment (`Guitariste`, `Guitare` et les interfaces)
- Configurer **Spring** dans l'application
- Faire jouer le guitariste !
 - Ne faire aucune instantiation!



ANNOTATIONS

DÉCLARATION PAR ANNOTATION

DÉCLARATION PAR ANNOTATION

- Déclaration d'un **bean**
 - Annotations
 - `@Configuration`
 - `@Component`
 - `@Controller / @RestController`
 - `@Service`
 - `@Repository`
- Injection d'un **bean** (instance managée par **Spring uniquement**)
 - `@Autowired` (avec `@Qualifier("nom")` possible)
 - `@Inject` (équivalent JSR330 de `@Autowired`)
 - `@Resource(name = "nom")`

DÉCLARATION PAR ANNOTATION

Annotation	JSR	Cas d'utilisation
@Component	@Named	Sauf @Configuration
@Qualifier	@Qualifier	
@Autowired	@Inject	@Inject est possible sur <i>static</i>
@Autowired + @Qualifier	@Resource(name = "nomBean")	

DÉCLARATION PAR ANNOTATION

- Annoter les classes
- Par défaut, le nom du *bean* sera le nom de la classe (avec une minuscule)
 - On peut modifier ce comportement en donnant une valeur aux annotations

Annotation	Définition	Cas d'utilisation
@Component	Composant Spring	Tous les cas, sauf ... (voir ci-dessous)
@Controller	Composant de type Controller	Point d'accès (comme les Servlets)
@RestController	Composant de type Controller	Point d'accès Service Web REST
@Repository	Composant de type Repository	Classe entrepôt (Repository par exemple)
@Service	Composant de type Service	Fournisseur de service
@Configuration	Classe de configuration	Configuration SPRING

DÉCLARATION PAR ANNOTATION

- Pour que la configuration de **Spring** voit les classes annotées
 - Il faut scanner les packages dans lesquels se trouvent ces classes
 - Ajouter l'annotation `@ComponentScan` à la configuration **Spring**

```
@Configuration  
@ComponentScan({ "fr.formation" })  
public class AppConfig {  
}
```

DÉCLARATION PAR ANNOTATION

- Pour injecter une référence gérée par **Spring**
 - Utilisation de l'annotation `@Autowired` sur une propriété
 - Pour que ce soit fonctionnel, il faut que toutes les références soient gérées par **Spring**

```
@Component
public class Guitariste implements IMusicien {
    @Autowired
    private IInstrument instrument;
}
```

- Ne fonctionnera que s'il y a un (et un seul) `IInstrument`
 - Annoté de `@Component`
 - Ou déclaré en tant que **bean** dans la configuration

DÉCLARATION PAR ANNOTATION

- `@Autowired` retrouve une dépendance
 - 1- Via le type de l'attribut ou de l'argument
 - 2- Via le nom de l'attribut ou de l'argument
 - Fonctionne très bien quand une seule référence est disponible, sinon :
 - Il faut utiliser `@Qualifier("nom")` pour retrouver la dépendance
 - Et/ou préciser `@Primary` (sur le bean) si on veut utiliser un **bean** spécifique en priorité

EXERCICE

- Remplacer la déclaration des **beans** en déclaration par annotation

EXERCICE

- Reprendre l'exercice sur les musiciens
- Cette fois-ci :
 - Choisir un instrument pour le musicien
 - Pour le guitariste : Guitare ou Ukulele
 - Pour le pianiste : Piano ou Synthé
 - Choisir qui doit jouer (le guitariste ou le pianiste)
 - Demander à l'utiliser de choisir le musicien qui doit jouer !
- Utiliser l'annotation `@Qualifier("nom")` avec `@Autowired`
 - Pour retrouver un **bean** par son nom



CYCLES DE VIE

LES SCOPES ET CYCLES DE VIE

CYCLES DE VIE

- En résumé
 - On déclare en tant que **bean** les dépendances par une approche déclarative (**XML ou JAVA**)
 - Toutes nos déclarations de dépendances sont au même endroit
 - **Spring** injectera les dépendances après l'instanciation des objets
 - Donc les références injectées ne sont pas disponibles dans le constructeur de l'objet

CYCLES DE VIE

- Au besoin, utiliser l'annotation `@PostConstruct` sur une méthode
 - Elle s'exécutera juste après l'instanciation !

```
@Component
public class Guitariste implements IMusicien {
    @Autowired
    private IInstrument instrument;

    public Guitariste() {
        this.instrument; //Pas dispo
    }

    @PostConstruct
    public void init() {
        this.instrument; //Disponible
    }
}
```

CYCLES DE VIE

- Ou utiliser `@Autowired` sur le constructeur
 - (avec un argument du type de la dépendance)
 - Dans ce cas, **Spring** cherchera à injecter la dépendance dès la construction de l'objet

```
@Component
public class Guitariste implements IMusicien {
    private IInstrument instrument;

    @Autowired
    public Guitariste(IInstrument instrument) {
        this.instrument = instrument;
    }
}
```

CYCLES DE VIE

- Ou utiliser `@Autowired` sur le setter
 - Fonctionne comme un `@Autowired` sur l'attribut
 - Permet d'avoir le setter à disposition

```
@Component
public class Guitariste implements IMusicien {
    private IInstrument instrument;

    @Autowired
    public void setInstrument(IInstrument instrument) {
        this.instrument = instrument;
    }
}
```

CYCLES DE VIE

- Par défaut, toutes les instances sont des instances **Singleton**
 - Créées au démarrage du contexte de **SPRING**
 - (et par extension, souvent au démarrage de l'application)
 - Scope « singleton »
- Possible de modifier ce comportement avec l'annotation `@Scope`

– prototype	<i>Une instance est créée à chaque demande</i>
– application	<i>Une instance est créée par contexte d'application (contexte Web)</i>
– request	<i>Une instance est créée par requête (contexte Web)</i>
– session	<i>Une instance est créée par session (contexte Web)</i>
– websocket	<i>Une instance est créée par websocket (contexte Web)</i>
– thread	<i>Une instance par thread</i>
- Si on souhaite rester en **Singleton**, mais empêcher l'instanciation au démarrage
 - Utilisation de l'annotation `@Lazy` sur la classe ou le **bean** déclaré



PROFILES

LES PROFILES DE CONFIGURATION

PROFILES

- Il est possible d'activer une configuration spécifique à un instant donné
 - Environnement de développement
 - Tests unitaires
 - Environnement de test
 - Environnement de production
 - Activer une journalisation
 - ...
- Ceci se fait via l'annotation `@Profile`, sur la classe de configuration, ou sur les **beans**

PROFILES

- Pour activer un profile, il suffit de démarrer l'application en spécifiant l'option
 - `-Dspring.profiles.active`, exemple :
 - `-Dspring.profiles.active=dev`
 - (Dans Eclipse, ajouter à VMArguments dans le configurateur des démarrages)



PROPERTIES

VARIABLES DANS FICHIER

CONFIGURATION AVEC PROPERTIES

- Mise en place d'une configuration avec un fichier `.properties` (clé = valeur)
 - Fichier `main/resources/musique.properties`

```
musique.instrument = guitare
```

```
musique.instrument.guitare.son = GLINK GLINK GLINK
```

CONFIGURATION AVEC PROPERTIES

- On précise à **Spring** d'aller récupérer ce fichier properties
- On manipule les propriétés avec une **SpEL (Spring Expression Language)** \${ }
- Sur l'attribut d'une classe

```
@Resource(name = "${musique.instrument}")  
private IInstrument instrument;
```

```
@Value("${musique.instrument.guitare.son}")  
private String instrumentNom;
```

CONFIGURATION AVEC PROPERTIES

- On précise à **Spring** le ou les fichiers properties avec l'annotation `@PropertySource`
 - Puis on injecte les propriétés dans un objet de type `Environment`, si nécessaire

```
@Configuration  
@PropertySource("classpath:/musique.properties")  
public class AppConfig {  
    @Autowired  
    private Environment env;  
  
    @Bean  
    public Guitariste guitariste() {  
        env.getProperty("musique.instrument")  
    }  
}
```

EXERCICE

- Mettre en place un fichier *.properties*
 - Préciser quel musicien doit jouer
 - Préciser le son de tous les instruments
 - Récupérer ces valeurs dans les instruments directement
 - Adapter le programme en conséquence



ASYNCHRONE

DES MÉTHODES ASYNCHRONES

ASYNCHRONE

- Annoter la classe de configuration de `@EnableAsync`
- Annoter la méthode asynchrone de `@Async`
 - ATTENTION
 - Ne fonctionne que sur les classes managées par **SPRING (beans)**
 - Ne fonctionne que lors d'un appel d'une classe externe (Classes Proxy)

EXERCICE

- Mettre en place une méthode asynchrone
 - Dans une classe de Service
 - Avec un temps de pause du Thread de quelques secondes
 - Et une impression de messages dans la console
- L'appeler depuis le programme principal



TÂCHES PLANIFIÉES

DES MÉTHODES PLANIFIÉES

TÂCHES PLANIFIÉES

- Annoter la classe de configuration de `@EnableScheduling`
- Annoter la méthode planifiée de `@Scheduled`
 - Ne fonctionne que sur les classes managées par **SPRING**
 - Nul besoin d'appeler cette méthode, **SPRING** le fait
 - La méthode ne retourne rien
- Options possibles
 - `initialDelay` Durée en millisecondes avant la première exécution
 - `fixedRate` Durée en millisecondes entre une exécution et la suivante
 - `fixedDelay` Durée en millisecondes entre la fin d'une exécution et le début la suivante
 - `cron` Expression Cron
`<minute> <heure> <jour-du-mois> <mois> <jour-de-la-semaine>`

EXERCICE

- Mettre en place une tâche planifiée
 - Dans une classe de Service
 - Imprime du texte dans la console toutes les 2 secondes