Exercices SBO

Il est demandé de créer un répertoire de travail (Workspace) nommé `WorkspaceSBO`. Tous les projets seront réalisés dans ce répertoire. Le choix d'utiliser l'outil d'automatisation de tâche `Maven` simplifiera la création des projets.

Exercice1

Il est demandé de faire une simple application Spring Boot pour afficher tous les beans chargés dans la configuration de base de l'application. Dans Boot Spring, vous on peut utiliser `appContext. getBeanDefinitionNames ()` pour obtenir tous les beans chargés par la configuration Spring Boot.

Etape 1 : création du projet

A partir d'Eclipse, il faut choisir `File -> New -> Project…` et choisir un `Maven Project.`

Choisir un projet parent Spring

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.1.0.RELEASE</version>
</parent>
```

Etape 2 : création d'une classe AppMain qui implémente CommandLineRunner

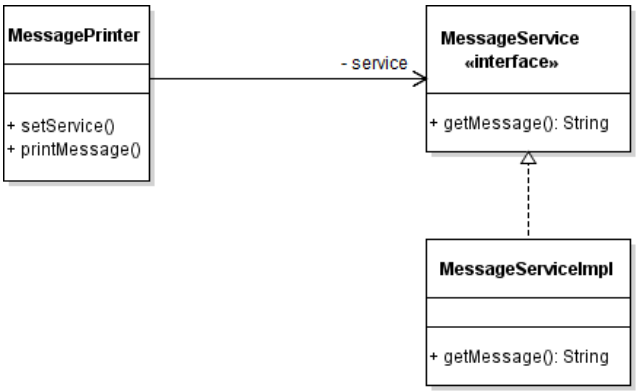
Cette classe AppMain aura :

- Un attribut de type ApplicationContext injecté par Spring en fonction de son type
- Une méthode main principale qui lance une SpringApplication par sa méthode run
- Une méthode run faisant l'inventaire des beans chargés dans la configuration Spring

Ajouter un fichier de configuration pour log4j.

Etape 3 : ajout de son propre service

Il est demandé d'ajouter un package service avec les classes suivantes :



MessagePrinter est annoté Component

MessageServiceImpl est annoté Service

Etape 4 : Contruire un livrable et une exécution

Il est important de déclencher les phases essentielles du cycle de vie d'un projet Maven qui sont :

- `mvn clean`
- `mvn install`

Le premier supprime tout ce qui aurait été généré depuis la dernière fabrication du livrable. Le second fournit un livrable ou fichier .jar dans le répertoire `target`. Pour déclencher ces deux phases dans cet ordre, il est demandé de sélectionner le projet puis grâce au menu contextuel de choisir `Run as -> Maven clean`, suivi de `Run as -> Maven install`.

Pour ce faire, sélectionner ce projet puis le menu `Run as -> Run Configurations …`

Ajouter le goal: `mvn spring-boot:run`

Affichage

```
application
customerRepository
customerRepositoryImpl
dataSource
dataSourceInitializedPublisher
dataSourceInitializer
dataSourceInitializerPostProcessor
emBeanDefinitionRegistrarPostProcessor
entityManagerFactory
entityManagerFactoryBuilder
hikariPoolDataSourceMetadataProvider
jdbcTemplate
jpaContext
//...
```

Exercice2

Il est demandé de faire une application Spring Boot pour charger les propriétés. L'annotaton Spring Boot @ConfigurationProperties permet de laisser au développeur le soin d'associer un fichier de propriété avec un objet facilement.

Etape 1 : création du projet par Initializer

Avec un navigateur, accéder au site `https://start.spring.io`

Configurer un projet pour faire une application Spring Boot web avec le module de test. Puis générer le projet.

Copier le fichier zip obtenu dans votre workspace et dezipper le.

A partir d'Eclipse, il faut choisir `File -> Import -> Project…` et choisir un `Maven Project`.

Ensuite, il est nécessaire de mettre à jour les dépendances Maven (mvn dependency :resolve)

Etape 2 : création des fichiers de ressources

Normalement, vous utilisez l'annotation @Value pour injecter une valeur issue d'un fichier .properties. Cela est utile pour les fichiers .properties petits et simples. Il est demandé de créer un fichier `global.properties` dans le répertoire `src/main/resources`

```
email=vincent.time@paris.fr
thread-pool=20
```

Puis dans un package `exercice2`, créer une classe `GlobalProperties`

```
@Component
@PropertySource("classpath:global.properties")
public class GlobalProperties {

    @Value("${thread-pool}")
    private int threadPool;

    @Value("${email}")
    private String email;

    //getters and setters
}
```

Enfin créer une classe annotée @Controller pour utiliser cette propriété dans une méthode annotée @RequestMapping

```
@Controller
public class Exercice2Controller {

    private static final Logger logger =
LoggerFactory.getLogger(Exercice2Controller.class);

    private GlobalProperties global;
```

```
    @Autowired
    public void setGlobal(GlobalProperties global) {
        this.global = global;
    }

    @RequestMapping("/")
    public String welcome(Map<String, Object> model) {

        String globalProperties = global.toString();

        logger.debug("Welcome {}, {}", global);

        model.put("message", globalProperties);
        return "welcome";
    }
}
```

Enfin construire une class `AppMain` pour démarrer l'application web

```
@SpringBootApplication
public class SpringBootWebApplication {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(SpringBootWebApplication.class, args);
    }

}
```

Etape 4 : Construire un livrable et une exécution

Il est important de déclencher les phases essentielles du cycle de vie d'un projet Maven qui sont :

- `mvn clean`
- `mvn install`

Démarrer Spring boot avec mvn spring-boot:run

Par un navigateur accéder à l'url de base / pour invoquer la méthode RequestMapping sur /.

Etape 5 : Construire classe annoté ConfigurationProperties

Modifier la classe `GlobalProperties` afin d'utiliser l'annotation @ConfigurationProperties

```
import org.springframework.boot.context.properties.ConfigurationProperties;

@Component
@PropertySource("classpath:global.properties")
@ConfigurationProperties
public class GlobalProperties {
    private int threadPool;
    private String email;

    //getters and setters

}
```

Les noms des attributs suffisent à faire le mapping

Refaire une exécution mvn spring-boot:run

Et un test Web

Etape 6 : Construire un fichier plus complexe

Définir un fichier de propriétés plus complexe du type suivant : application.properties

```
#Logging
logging.level.org.springframework.web=ERROR
logging.level.com.paris=DEBUG

#Global
email=vincent.time@paris.fr
thread-pool=20

#App
app.menus[0].title=Home
app.menus[0].name=Home
app.menus[0].path=/
app.menus[1].title=Login
app.menus[1].name=Login
app.menus[1].path=/login

app.compiler.timeout=5
app.compiler.output-folder=/temp/

app.error=/error/
```

Puis créer un bean @ConfigurationProperties pour faire le mapping de tout le fichier

```
@Component
@ConfigurationProperties("app") // prefix app, find app.* values
public class AppProperties {

    private String error;
    private List<Menu> menus = new ArrayList<>();
    private Compiler compiler = new Compiler();

    public static class Menu {
        private String name;
        private String path;
        private String title;

        //getters and setters

        @Override
        public String toString() {
            return "Menu{" +
                    "name='" + name + '\'' +
                    ", path='" + path + '\'' +
                    ", title='" + title + '\'' +
                    '}';
        }
    }
```

```
    public static class Compiler {
        private String timeout;
        private String outputFolder;

        //getters and setters

        @Override
        public String toString() {
            return "Compiler{" +
                    "timeout='" + timeout + '\'' +
                    ", outputFolder='" + outputFolder + '\'' +
                    '}';
        }

    }

    //getters and setters
}
```

Retirer l'annotation @PropertySource("classpath:global.properties") de la classe GlobalProperties.

Modifier la classe Exercice2Controller afin qu'elle prenne en compte les deux properties :

```
@Controller
public class Exercice2Controller {

    private static final Logger logger =
LoggerFactory.getLogger(Exercice2Controller.class);

    private AppProperties app;
    private GlobalProperties global;

    @Autowired
    public void setApp(AppProperties app) {
        this.app = app;
    }

    @Autowired
    public void setGlobal(GlobalProperties global) {
        this.global = global;
    }

    @RequestMapping("/")
    public String welcome(Map<String, Object> model) {

        String appProperties = app.toString();
        String globalProperties = global.toString();

        logger.debug("Welcome {}, {}", app, global);

        model.put("message", appProperties + globalProperties);
        return "welcome";
    }
}
```

Refaire une exécution mvn spring-boot:run

Et un test Web

Etape 7 : Construire un fichier YAML

Il est important de renommer le fichier de propriétés standard afin qu'il n'y ait pas de confusion.
Créer un fichier application.yml

```
logging:
  level:
    org.springframework.web: ERROR
    com.mkyong: DEBUG
email: vincent.time@paris.fr
thread-pool: 20
app:
  menus:
    - title: Home
      name: Home
      path: /
    - title: Login
      name: Login
      path: /login
  compiler:
    timeout: 5
    output-folder: /temp/
  error: /error/
```

Refaire une exécution mvn spring-boot:run

Et un test Web

Etape 7 : Validation

L'annotation @ConfigurationProperties supporte JSR-303 Bean validation-javax.

Modifier la classe `GlobalProperties` afin d'ajouter des contraintes sur les attributs

```
@Component
@ConfigurationProperties
public class GlobalProperties {

    @Max(5)
    @Min(0)
    private int threadPool;

    @NotEmpty
    private String email;

    //getters and setters
}
```

Démarrer Spring boot à nouveau, et revoir la console suite à une requête sur /

Exercice3

Il est demandé de faire une application Spring Boot pour charger les propriétés au format YAML ou properties dans les séquences suivantes :

- `application-{profile}.{properties|yml}`
- `application.{properties|yml}`

Etape 1 : création du projet

A partir d'Eclipse, il faut choisir `File -> New -> Project…` et choisir un `Maven Project.`

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

Spring Boot Exercices

1- Spring boot et log
2- Spring boot et service REST
3- Spring Goot et JPA