Principes et Commandes en Java pour Spring - Srping Boot

Spring

Principe :

Dans le dossier src/test/, on créé une classe miroir pour chacune des classes de notre projet afin de pratiquer des tests unitaires sur chacune des méthodes qu'elles contiennent.

Une classe métier => une classe test.

Une méthode => une ou plusieurs méthodes contenant un test unitaire vérifiant une condition particulière.

Permet de pratiquer des tests avant d'intégrer le code au projet et de corriger et consolider au maximum le code de chaque méthode afin d'éviter les bugs, les exceptions non gérées, …

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Symbole | Signification | Code | | Utilisation | | |
| Spring |  | | | | | |
|  |  |  |  | | |  |
|  |  |  |  | | |  |
|  |  |  |  | | |  |
| Annotations |  |  |  | | |  |
| @Component | générique pour identifier un composant (= bean Spring) géré par Spring |  |  | | |  |
| @Service | identifie les beans contenant le code métier (nécessitant une ou plusieurs injections) |  |  | | |  |
| @Repository | identifie les beans liés à la persistence des données (DB, ...) |  |  | | |  |
| @Controller | identifi les points d'entrées de l'appli |  |  | | |  |
| @Primary | à ajouter après l'une des annotations précédentes pour définir le bean à utiliser par défaut en cas de création de plusieurs beans de même type  ou ajout d'un identifiant  Rq: par défaut, l'identifiant = le nom de la classe avec la première lettre en minuscule. | @Service  @Primary  public class FixedMeasure implements Measure {  @Service  public class VariableMeasure implements Measure {  @Service("fixedMeasure")  public class FixedMeasure implements Measure {  @Service("variableMeasure")  public class VariableMeasure implements Measure { |  | | |  |
| @Configuration | identifie les classe ApplicationConfig (= Factory) qui vont gérer les beans (injection à la demande, persistance, ...) | @ComponentScan  @Comfiguration  public class MonApplicationConfig {  } |  | | |  |
| @ComponentScan | autoscan des éléments de l'appli afin de repérer automatiquement les Beans sans avoir à les déclarer dans la Factory |  | | | ATTENTION : ne scanne que les classe contenue dans le même package (et sous-packages) de la classe en question |
| @ComponentScan("com.monapp.monpackage") | scanne les classes du package et sous-package indiqué | @ComponentScan("com.monapp.monpackage")  @Comfiguration  public class MonApplicationConfig {  ApplicationContext context = new AnnotationConfigApplicationContext(MonApplicationConfig.class);  MaClasseService maClasseService = context.getBean(MaClasseService.class);  } | | | |  |
| ApplicationContext | objet servant à obtenir les bean manuellements |  |
| @Autowired | permet d'injecter un bean géré par Spring dans une classe  bean non trouvé  plusieurs beans peuvent correspondre au type demandé | *Injection par setter :*  @Component  public class MaClasseImpl implements MonInterfaceService {  @Autowired  private MonAutreClasseImpl maci;  }  @Component  public class MaClasseImpl implements MonInterfaceService {  private MonAutreClasseImpl maci;  @Autowired  public void setMonAutreClasseImpl(MonAutreClasseImpl maci) {  this.maci = maci;  }  NoSuchBeanDefinitionException  NoUniqueBeanDefinitionException | *Injection par constructeur :*  @Component  public class MaClasseImpl implements MonInterfaceService {  private MonAutreClasseImpl maci;  @Autowired  public MaClasseImpl(MonAutreClasseImpl maci) {  this.maci = maci;  } | | | Rq: recherche le bean à injecter d'abord par type, puis par nom (identifiant)  Rq: @Autowired est facultatif pour l'injection par constructeur à condition qu'il n'existe qu'un seul constructeur ! |
| (requires = "false") | injection facultative (si la classe bean n'est pas trouvée, injecte null et ne retourne pas l'erreur NoSuchBeanDefinitionException) | @Autowired(requires = "false") |  | | |  |
| @Qualifier(" ") | définit le bean à utiliser via son identifiant en cas d'ambiguïté (si plusieurs beans sont possibles car de mêmes types => voir @Primay) | *Injection par setter :*  @Component  public class MaClasseImpl implements MonInterfaceService {  @Autowired  @Qualifier("maciA")  private MonAutreClasseImpl maciA;  } | *Injection par constructeur :*  @Component  public class MaClasseImpl implements MonInterfaceService {  private MonAutreClasseImpl maci;  @Autowired  public MaClasseImpl(@Qualifier("maciB") MonAutreClasseImpl maciB) {  this.maci = maci;  } | | |  |
| @Ressource | équivalente à @Autowired, mais effectu la recherche du bean à injecter d'abord par nom, puis pas type. |  |  | | |  |
| @Scope( ) | définit le scope du bean | @Service  @Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)  @Lazy  public class MaClasseService {  } | .SINGLETON  .SCOPE\_PROTOTYPE  .REQUEST  .SESSION  .APPLICATION  .WEBSOCKET | | | (défaut) une seule instance pour toute l'appli  une instance à chaque injection  une instance par requête  une instance par session utilisateur (dépréciée)  une instance par application  une instance par socket |
| @Lazy | instancié et injecté uniquement lorsque il y en a besoin => allège le lancement de l'appli  (par défaut, tout est instancié à l'ouverture de l'appli) |  | | |  |
|  |  |  |  | | |  |
| Resources | données de fichiers externes ou de .properties |  |  | | |  |
|  | **Ressource externe** |  | | | |  |
| ResourceLoader | bean de Spring qui permet d'accéder aux ressource (classpath, file, url) | ressourceLoader.getRessource("classpath:monFichier.txt");  ressourceLoader.getRessource("file:///C:/monChemin/monFichier.txt");  ressourceLoader.getRessource("url:https://www.monsite.com");  @Component  public class MaClasseService {  @Autowired  private RessourceLoader ressourceLoader;  public void readFile() {  try {  Ressource ressource = ressourceLoader.getRessource("file:///C:/monChemin/monFichier.txt");  InputStream stream = resource.getInputStream();  Scanner scanner = new Scanner(stream).useDelimiter("\\n");  while (scanner.hasNext()) {  System.out.println(scanner.next());  }catch (IOException e) {  }  }  } | | | |  |
| .getRessource( ) | sélectionne le type de ressource et son adresse | ATTENTION : ///C:/ nécessaire qu'avec Windows |
|  | Fichier .properties |  |  | | |  |
| .properties | fichiers de config dans ***src/main/ressources/*** | *contenu de* **monFichierConfig.properties** *:*  myapp.name=BigCorp application  myapp.version=1  myapp.emails=contact-dev@bigcorp.com,test@bigcorp.com  myapp.baseuri=${BASE\_URI:http://localhost:8080} | | | => List ou Set  => valeur de la variable environnement, si absente : valeur par défaut |  |
|  | Propriétés |  |  | | |  |
| @PropertySource( ) | charge le fichier .properties indiqué en paramètre | @ComponentScan  @Configuration  @PropertySource("classpath:monFichierConfig.properties")  public class MonApplicationConfig {  FileNotFoundException |  | | |  |
|  | sans lever d'exception  en cas de fichier non trouvé : | @PropertySource(value = "classpath:monFichierConfig.properties" ignoreRessourceNotFound=true) | | | |  |
| @Value( ) | récupère la valeur de la propriété dans le fichier | @Value("${myapp.name}")  private String name  @Value("${myapp.version}")  private Integer version  @Value("${myapp.name}")  private Set<String> emails; |  | | |  |
|  |  |  |  | | |  |
|  | Profils |  |  | | |  |
| @Profil( ) | défini le profil pour lequel la classe de configuration (ApplicationConfig) ou le bean va être utilisé | @ComponentScan  @Comfiguration  public class monApplicationConfig {  }  @Configuration  @Profile("!prod") *//ou* @Profile("default")  public class monApplicationConfig {  }  @Configuration  @Profile("prod")  public class monApplicationConfig {  } |  | | | ATTENTION : si elle est utilisée sur la classe, @Autoscan ne doit pas être appliquée car il ne peut être utilisé qu'une seule fois : sur la classe ApplicationConfig générale |
| .setActiveProfiles( ) | utilisation d'un profil via l'ApplicationContext | AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();  context.getEnvironment().setActiveProfiles("dev", "cloud");  context.register(BigCorpApplicationConfig.class);  context.refresh(); | | | | Rq: 3 méthodes d'activation du profil :   1. manuellement via la variable d'environnement de Spring :   Edit Configurations > Environment variables  -> spring.profiles.active=prod   1. en ligne de commande   Dspring.profies.active="prod, cloud"   1. dans l'appli avec .setActivateProfils( ) |
|  | Environnement |  |  | | |  |
| Environment |  | @Component  public class MaClasseService {  @Autowired  private Environment environment;  public ApplicationInfo applicationInfo() {  String name = environment.getRequiredProperty("myapp.name");  Integer version = environment.getRequiredProperty("myapp.version", Integer.class);  Set<String> emails = environment.getRequiredProperty("myapp.emails", Set.class);  String javaHome = environment.getRequirdProperty("JAVA\_HOME");  return new ApplicationInfo(name, version, emails);  }  } | | | |  |
|  |  |  |  | | |  |
| AOP | AOP = Aspect-Oriented Programming | permet de surcharger une méthode en ajoutant des comportements  Pour cela, on localise les méthodes à modifier (= les join points) (plusieurs join points = point cut)  on définit le code à rajouter avant et/ou après le résultat de cette méthode (les Advice)  L'ensemble est appelé : Aspect | | | |  |
| Proxy | on peut recréer ce comportement manuellement | public class ComputerProxy extends Computer {  public int add(int a, int b) {  System.out.println("DEBUT Appel de la méthode add");  int result = super.add(a, b);  System.out.println("FIN Appel de la méthode add : " + result)  return result;  }  } | | | |  |
| build.gradle | dépendances à ajouter dans **build.gradle** | compile "org.springframework:spring-aop:${springVersion}  compile "org.aspectj:aspectjweaver:1.8.13" | | | | Rq: AspectJ est l'implémentation de référence, mais Spring AOP (qui se base sur les bibliothèques AscpectJ) est plus simple à utiliser |
| @EnableAspectJAutoProxy | indique à la classe ApplicationConfig de gérer également l'AOP | @Configuration  @EnableAspectJAutoProxy  public class MonApplicationConfig {  } |  | | | ATTENTION : ne doit être appliqué que sur des classes implémentant une interface ! Sinon rend la déclaration plus complexe. |
| @ | définition d'une nouvelle annotation spécifique pour chaque AOP | @Target(ElementType.METHOD)  @Retention(RetentionPolicy.RUNTIME)  @Documented  public @interface MonAnnotationAOP { } |  | | |  |
| @MonAnnotationAOP  @Aspect | annoter les méthodes concernées avec l'annotation personnalisée  définit l'Advice, cad la surcharge autour de la méthode | @Service  public class MaClasseServiceImpl implements MaClasseService {  @MonAnnotationAOP  @Override  public String maMethode(String monParametre) {  }  }  @Component  @Aspect  public class MonAnnotationAOPAspect {  **@Before("@annotation(**MonAnnotationAOP**)")**  public void logServiceBeforeCall(JoinPoint jp) {  System.out.println("Le bean concerné par l'Aspect est : " + jp.target());  System.out.println("Appel de la méthode " + jp.getSignature());  System.out.println("avec les agruments :" + jp.args());  }  *//permet de choisir si la méthode est exécutée ou non !*  **@Around("@annotation(**com.training.spring.bigcorp.config.Secure**)")**  public Object doCheckSecurity(ProceedingJoinPoint pjp) throws Throwable Object returnedValue;  CurrentUser user = applicationContext.getBean(CurrentUser.class);  if(securityService.checkAccess(user)){  return pjp.proceed();  }  throw new ForbiddenTargetException("User not allowed to use " + "this feature");  }  *//exécuté seulement si la méthode n'a pas levé d'exception*  **@AfterReturning(pointcut = "@annotation(**MonAnnotationAOP**)", returning = "element")**  public void logServiceAfterCall(JoinPoint jp, Object element) {  if (element == null) {  System.out.println("Finder " + jp.getTarget() + "returns null";  } else {  System.out.println("Finder " + jp.getTarget() + "returns "+ element.toString());  }  *//exécuté seulement si la méthode a retourné une exception*  **@AfterThrowing(pointcut = "@annotation(**Finder**)", throwing = "ex")**  public void logServiceAfterException(RuntimeException ex) {  System.out.println("Error " + ex.getMessage());  }  *//exécuté après la méthode, qu'il y ait eu exception ou non*  **@After("@annotation(**MonAnnotationAOP**) && args(monParametre)")**  public void logServiceBeforeCall(JoinPoint jp, String monParametre) {  System.out.println("Le paramètre qui a été utilisé par la méthode était : " + monParametre);  }  } | | | |  |
|  |  |  |  | | |  |
|  |  |  |  | | |  |
|  |  |  |  | | |  |
|  |  |  |  | | |  |
|  |  |  |  | | |  |
|  |  |  |  | | |  |

Spring Boot

Principe :

Spring Boot = projet qui embarque avec lui un server pour être déployé automatiquement dans le cloud

Utilise : framework Spring 5, Java 8, Tomcat 8.5 par défaut (compatible Jetty 9.4, Undertow 3.1)

Création d'un projet Spring Boot

Démarrer un projet sur https://start.spring.io/ (avec les dépendances Spring Web Starter, Spring Boot Devtools) puis ouvrir le projet avec l'option auto-import sur InteliJ

Fichiers .properties

application.properties fichier contenu dans ***src/main/resources/*** chargé automatiquement (inutile d'utiliser @PropertySource) quelque soit le profil

application-default.properties idem

application-prod.properties fichiers contenant les valeurs des propriétés spécifiques à chaque profil et venant surcharger les propriétés par défaut si le profil est activé

application-cloud.properties

…

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Symbole | Signification | Code | | Utilisation | |
| Spring Boot |  | | | | |
| @SpringBootApplication | méta-annotation remplacent @Configuration + @ComponentScan + @EnableAutoConfiguration + @EnableConfigurationProperties  => autoscan Spring sur tout le projet pour gérer l'injection des Beans, permettre l'auto-configuration, permet de récupérer les valeurs des propriétés dans les fichiers *applications.properties* et de les injecter dans les Beans dédiés annotés avec @ConfigurationProperties(prefix = "...")  *(voir plus bas :* .properties*)* | @SpringBootApplication  @EnableConfigurationProperties  public class MonApplication {  public static void main(String[] args) {  }  } | |  |  |
|  |  |  | |  |  |
| Tests |  |  | |  |  |
| @RunWith(SpringRunner.class) | à placer sur les classes Test afin d'indiquer que le test est géré par Spring  (injection, etc…) | @RunWith(SpringRunner.class)  @SpringBootTest  public class MaClasseTest {  @Test  public void testMaMehtode() {  }  } | |  |  |
| @SpringBootTest | à placer sur les classes Test afin de créer un application context spécifique au test |  |  |
| Mockito |  |  | |  |  |
| @RunWith(MockitoJUnitRunner.class) | à placer sur les classes de tests utilisant Mockito afin d'injecter les mock définis par @Mock automatiquement | @RunWith(MockitoJUnitRunner.class)  public class MaClasseTest {  @Mock  MonInterface monMock;  @Test  public void testMaMethode() {  MaClasse maClasse = new MaClasse();  MonObjet bouchon = new MonObjet();  Mockito.doReturn(new MonObjetRetour()).when(monMock).maMethodeMockee();  maClasse.monImplement = monMock;  ...  Mockito.verify(monMock).maMethodeMockee();  } | | |  |
| @Mock |  |  |
|  |  |  | |  |  |
| Ligne de commande |  |  | |  |  |
| clean assemble | crée le fichier .jar dans *build/libs/* si tous les tests unitaires sont passants | gradlew clean assemble | |  |  |
| java -jar .jar | lanche l'appli sur le serveur | java -jar build/libs/monProjet-0.0.1-SNAPSHOT.jar | |  |  |
| --continuous bootRun | lance l'appli sur le serveur en continue (prenant en compte les modif en temps réels)  (ctrl+C pour l'arréter) | gradlew --continuous bootRun | |  |  |
|  |  |  | |  |  |
| .properties |  |  | |  |  |
| @ConfigurationProperties(prefix = "…")  @Nested | permet de récupérer automatiquement les valeurs des propriétés et de les placer dans un bean sans avoir à utiliser @Autowired Environment ni @Value("${monappli.mapropriete}")  @Nested identifie la classe automatiquement sans avoir besoin de l'annoter avec @Component | @Component  @ConfigurationProperties(prefix = "**monappli**")  public class MonAppliProperties {  private String name;  private int version;  private Set<String> emails;  @NestedConfigurationProperty  private **MonSousGroupeDeProprietes** **sousgroupe**;  *// getter et setter*  }  \_\_\_  public class **MonSousGroupeDeProprietes** {  private int measureFixe;  private int measureVariable;    *//getter et setter*  } | | *Contenu du fichier* application.properties *:*  **monappli**.name=Mon nom  **monappli**.version=1.0.0  **monappli**.emails=contact@bigcorp.com, test@bigcorp.com  **monappli**.**sousgroupe**.measure-fixe=10  **monappli**.**sousgroupe**.measure-variable=20 | ATTENTION : pour les classes Test, ne fonctionne que si la classe principale (contenant la méthode main) est annotée avec @EnableConfigurationProperties ! |
|  |  |  | |  |  |
| Log | Spring Boot integre SLF4J et LogBack pour gérer les logs  Simplement besoin de définir les niveaux dans application.properties et d'appeler le LOGGER dans la classe | | |  |  |
| application.properties | définition des Logs | logging.file=c:/data/logs/monappli.log  logging.level.root=INFO;  logging.level.monpackage=ERROR  logging.level.monpackage.monsouspackage=DEBUG | | => fichier de sortie des Logs  => niveau de Log par défaut  => niveau de log spécifique aux classes d'un package et de ses sous-packages |  |
| LoggerFactory |  | @Service  public class MaClasseImpl implements MaClasse {  private final static Logger LOGGER = LoggerFactory.getLogger(MaClasseImpl.class);  public void maMethode(String maString) {  LOGGER.info("Appel de maMethode avec argument {}", maString);  ...  }  } | | |  |
| logback.xml | il est toujours possible de définir les propriétés des logs dans *src/main/resources/logback.xml*, utile notamment pour les system plus complexe de rolling | <configuration>  <include resource="org/springframework/boot/logging/logback/base.xml"/> (1)  <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender"> (2)  <file>/data/logs/bigcorp.log</file>  <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">  <FileNamePattern>/data/logs/bigcorp/recovery.%d{yyyy-MM-dd}.%i.zip</FileNamePattern>  <maxFileSize>32MB</maxFileSize>  <maxHistory>15</maxHistory>  <totalSizeCap>500MB</totalSizeCap>  </rollingPolicy>  <encoder>  <pattern>${CONSOLE\_LOG\_PATTERN}</pattern> (3)  </encoder>  </appender>  <logger name="com.training.spring.bigcorp" level="DEBUG" additivity="false"> (4)  <appender-ref ref="FILE"/>  <appender-ref ref="CONSOLE"/>  </logger>  <root level="INFO"> (5)  <appender-ref ref="FILE" />  </root>  </configuration> | | | (1) inclusion de la configuration de Spring Boot (permet d'avoir accés au logger de base CONSOLE) (3)  (2) rolling, 32 Mo par fichier max, 15 fichiers max, 500 Mo max pour la taille de tous les fichiers réunis  (4) possibilité de définir plusieurs niveau par logger  (5) root level = niveau de log par défaut |
|  |  |  | |  |  |
| JPA |  |  | |  |  |
| build.gradle | ajout des dépendances : | compile('org.springframework.boot:spring-boot-starter-data-jpa')  compile('org.springframework.boot:spring-boot-starter-web')  compile('com.h2database:h2') | | => ajout de data-JPA  => permet accès de la DB via navigateur web  => Embarquement d'une DB (ici : H2) dans le projet Gradle |  |
| application.properties | configuration des propriétés : | # Spring boot : datasource  spring.datasource.url=jdbc:h2:mem:bigcorp;DB\_CLOSE\_DELAY=-1;DB\_CLOSE\_ON\_EXIT=FALSE  spring.datasource.platform=h2  spring.datasource.username=sa  spring.datasource.password=  spring.datasource.driverClassName=org.h2.Driver  spring.datasource.sqlScriptEncoding=UTF-8  # Spring boot : H2 datasource  spring.h2.console.enabled=true  spring.h2.console.path=/console  # Spring boot : JPA  spring.jpa.database-platform=org.hibernate.dialect.H2Dialect  spring.jpa.ddl-auto=create-drop  spring.jpa.show\_sql=true ----------------------------> | | => config de la DB  => permet accès de la DB via navigateur web  (***http://localhost:8080/console***)  => permet la création automatique des tables à partir des annotations indiquées dans les @Entity  => affiche les requetes SQL dans la console |  |
| schema.sql | fichier d'initialisation de la DB  ***src/main/resources/schema.sql***  (utile pour JDBC, mais facultatif si les tables sont auto-générées par JPA) | *Exemple :*  CREATE TABLE SITE(ID VARCHAR(255) PRIMARY KEY, NAME VARCHAR(255) NOT NULL);  CREATE TABLE CAPTOR(ID VARCHAR(255) PRIMARY KEY, NAME VARCHAR(255) NOT NULL,  SITE\_ID VARCHAR(255) NOT NULL);  ALTER TABLE CAPTOR ADD FOREIGN KEY (SITE\_ID) REFERENCES SITE(ID); | | |  |
| data.sql | fichier de remplissage de la DB  ***src/main/resources/data.sql*** | *Exemple :*  INSERT INTO SITE (ID, NAME, version) VALUES('site1', 'Bigcorp Lyon', 0);  INSERT INTO CAPTOR (DTYPE, ID, NAME, site\_id, default\_power\_in\_watt, version) VALUES('FIXED', 'c1', 'Eolienne', 'site1', 1000000, 0); | | |  |
| @DataJpaTest | à ajouter sur la classe Test pour utiliser JPA et le lien vers la DB | **@RunWith(SpringRunner.class)**  **@SpringBootTest**  @DataJpaTest  public class MaCLasseTest { | |  |  |
|  |  |  | |  |  |
| Data JPA | (Repository = DAO = Data Acces Object) |  | |  |  |
| JpaRepository< , > | interface dont doit hériter tous les DAO et permettant de ne pas avoir ré-écrire les méthodes principales (.findAll(), .save(), .remove(), ...) | public interface MonElementDao extends JpaRepository<MonElement, String> {  } | |  | ATTENTION : le second type générique est celui de l'id de l'élément (Integer, Long, String, ...) et servira pour les méthodes : .findById( ) |
| CustomDao | personnalisation de l'interface DAO   * en implémentant une autre interface personnalisée | public interface MonElementDao extends JpaRepository<MonElement, String>, MonElementCustomDao {  } | | public interface MonElementCustomDao {  public List<Captor> findBySiteId(String siteId);  } |  |
|  |  | @Repository  public class MonElementDaoImpl implements MonElementDao {  @PersistenceContext  private EntityManager em;  @Override  public List<Captor> findBySiteId(String siteId) {  return em.createQuery("select c from Captor c inner join c.site s where s.id = :siteId", Captor.class)  .setParameter("siteId", siteId)  .getResultList();  }  } | | |  |
| @Query  @Modifying | personnalisation de l'interface DAO   * en modifiant l'interface MonElementDao elle-même en utilisant les conventions de JpaRepository   @Query permet de définir une requête JPQL  @Modifying permet de modifier le comportement par défaut d'une requête automatique | public interface CaptorDao extends JpaRepository<Captor, String> {    List<Captor> findBySiteId(String siteId);  List<Captor> findByNameAndSiteId(String name, String siteId);  List<Captor> findByNameOrSiteId(String name, String siteId);  List<Captor> findDistinctBySiteId(String siteId);  List<Captor> countByName(String name, String siteId);  @Query("select c from Captor c where c.name=:name") ------------------------------------------------------>  Captor findByName(@Param("name") String name);    @Modifying  @Query("delete from Captor c where c.name = ?1")  void deleteByName(String name);  } | | | Rq: ici on aurait pu utiliser la gestion automatique comme les 5 exemples juste au dessus |
|  |  |  | |  |  |
| Annotation JSON | contrôle la serialisation/deserialisation si objet assicié à une API Rest |  | |  |  |
| @JsonProperty(" ") | permet d'indiquer le nom de la propriété associé au champ lors de la serialisation/déserialisation en JSON | public static class Imprimante implements Serializable {  @JsonProperty("nom\_imprimante")  private nom;  @JsonProperty("code\_imprimante")  private String code;  } | | *=> JSON associé :*  {  "nom\_imprimante": "Imprimante commune 1",  "code\_imprimante": "EPS21"  } |  |
|  |  |  | |  |  |
| Annotations de validation | contrôle que les valeurs des champs avant la persistance de l'élément |  | |  |  |
| @Version | active le controle optimiste de la concurrence, cad lorsque l'utilisateur manipule une entité, s'il essai de la persister mais que son numéro de version a changé entre temps, alors une exception OptimisticLockException est levée | @Entity  public class MonElement {  @Id  private String id = UUID.randomUUID().toString();  @Version  private Integer version;  ...  *//constructeur vide*  *//getter et setter*  }  OptimisticLockException | |  |  |
| @Null  @NotNull | peut-être null  ne peut pas être null | @Entity  public class Measure {  @Id  @GeneratedValue  private Long id;  @NotNull  @Past  private Instant instant;  @Min(0)  @NotNull  private Integer minPower;  @Min(0)  @NotNull  private Integer maxPower;  @Size(min = 15, max = 150)  @Email  private String email;  @AssertTrue(message = "minPower should be less than maxPower")  public boolean isValid() {  return this.minPower <= this.maxPower;  }  *// ...*  } | |  |  |
| @Min  @Max | valeur min pour un entier  valeur max pour un entier |  |  |
| @DecimalMin  @DecimalMax | valeur min pour un réel  valeur max pour un réel |  |  |
| @Digit | ne doit pas avoir plus de chiffres significatifs (av et ap la virgule) qu'indiqué |  |  |
| @Size  @Pattern | taille max pour une chaine de caractères  vérifie la conformité d'une chaine avec une RegEx |  |  |
| @Email | doit être un e-mail valide |  |  |
| @Future  @Past | la date doit être postérieure  la date doit être antérieure |  |  |
| @AssertTrue  @AssertFalse | la valeur doit être true  la valeur doit être false |  |  |
| @Valid | se place sur une prop pour la validée | *validation simplifiée d'un bean en utilisant les annotations de ses champs :*  Rq: *après la création du bean (déserialisation depuis client REST par exemple), il faut appeler monBean.isValid(); en capturant l'erreur possible.*  public void isValid() throws MonException {  *// Valide les champs selon leurs annotations*  ValidatorFactory factory = Validation.buildDefaultValidatorFactory();  Validator validator = factory.getValidator();  Set<ConstraintViolation<Measure>> violations = validator.validate(this);  StringBuilder errorMsgs = new StringBuilder();  for(ConstraintViolation<MonBean> violation : violations) {  errorMsgs.append("\n\t").append(violation.getMessage());  }  if(!violations.isEmpty()) {  throw new MonException(String.format("Erreur de validation de l'objet Measure. Violation des régles : %s", errorMsgs.toString()));  }  *//validation selon une règle personnalisée*  *//ex: si champ1 == X alors champs2 doit être == Y sinon, new MonException ou ajout à la liste errorMsgs*  *}* | |
| Test | exemple de classe Test avec les annotations de validations  Rq: Tests réalisés avec AssertJ | @Test  public void createShouldThrowExceptionWhenNameIsNull() {  Assertions.assertThatThrownBy(() -> {  measureDao.save(new Measure(null, null));  entityManager.flush();  })  .isExactlyInstanceOf(javax.validation.ConstraintViolationException.class)  .hasMessageContaining("must not be null");  }  @Test  public void createSimulatedCaptorShouldThrowExceptionWhenMinMaxAreInvalid() {  Assertions.assertThatThrownBy(() -> {  measureDao.save(new Measure(50, 25));  entityManager.flush();  })  .isExactlyInstanceOf(javax.validation.ConstraintViolationException.class)  .hasMessageContaining("minPower should be less than maxPower");  }  @Test  public void createShouldThrowExceptionWhenNameSizeIsInvalid() {  Assertions.assertThatThrownBy(() -> {  measureDao.save(new Measure("mail@sfr.fr"));  entityManager.flush();  })  .isExactlyInstanceOf(javax.validation.ConstraintViolationException.class)  .hasMessageContaining("size must be between 15 and 150");  } | | |  |
|  |  |  | |  |  |
| MVC |  |  | |  |  |
| build.gradle | importation de la dépendance  => application autonome, cad qui embarque son propre serveur auto configuré | dependencies {  implementation('org.springframework.boot:spring-boot-starter-web')  } | | |  |
| structure | resources web du projet à placer dans **src/main/resources** : | src/main/resources  /static  /css *=> fichiers CSS*  /img *=> fichiers images*  index.html  /template *=> vues partielles (.jsp, .mustache, ...)* | |  |  |
| @Controller | déclare un bean Spring de type Controller  equiv à une servlet, mappage via les annotations, redirection vers le modèle et la vue par ModelAndVue | @Controler  @RequestMapping("/suiteUrlA") *//gère tous les URL* **https://monAppli/suiteUrlA/...**  **@Transactional** ***//indispensable si entités liées à DB***  public class MonController {    @Autowired  MonElementDao monElementDao;  @RequestMapping(path = "/suiteUrlB", method = RequestMethod.GET)  public ModelAndView create() {  return new ModelAndView("maPage").addObject("monEntity", new MonEntity());  }  @GetMapping("/suiteUrlB/{entityId}")  public ModelAndView findById(@PathVariable String monId) {  MonElement monEntity = monEntityDao.findById(entityId).orElseThrow(IllegalArgumentException::new);  return new ModelAndView("maPageDeMonEntity").addObject("monEntity", monEntity);  }  @PostMapping(consumes = MediaType.APPLICATION\_FORM\_URLENCODED\_VALUE)  public ModelAndView save(MonEntity monEntity) {  entityDao.save(entity); ----------------------------------------------------------------------------------->  return new ModelAndView("monAutrePage");  }  @PostMapping("/suiteUrlB/delete?entityId=monId")  public ModelAndView delete(@RequestParam String entityId) {  Entity entity = entityDao.findById(entityId).orElseThrow(IllegalArgumentException::new);  entityDao.delete(entity);  return new ModelAndView("monAutrePage");  }  } | | | => génération automatique de l'entité via les paramètres du formulaire si ceux-ci respectent les noms des champs de l'entité |
|  | options des annotations :  défini le type de données acceptées en entrée  défini le type de données acceptées en sortie  filtre le mapping en fonction des attributs dans le head de la requete html  filtre le mapping en fonction des paramètres de l'URL | | @PostMapping(**consumes** = MediaType.APPLICATION\_FORM\_URLENCODED\_VALUE)  @GetMapping(**produces** = MediaType.APPLICATION\_XML\_VALUE)  **headers**  **params** | |  |
|  | objet récupérable en paramètre dans les méthodes annotées @XxxMaping( ) :  accés au modele (map clé/valeur)  objet contenant les infos régionales de l'utilisateur envoyées par http  accés à l'objet session  accès à l'objet request et info valides seulement le temps de la requête | | Model  Locate  HttpSession  HttpServletRequest |  |  |
|  |  |  | |  |  |
| Mustache | importation de la dépendance dans **build.gradle**, permet d'utiliser les fichiers .mustache | dependencies {  implementation('org.springframework.boot:spring-boot-starter-mustache')  } | | |  |
| .mustache | équiv de .jsp en plus basique mais plus performant | <header>  {{**>**partial-header}}  </header>  <section>  {{**#**items}}  Name: {{name}}  Price: {{price}}  {{#features}}  Feature: {{description}}  {{/features}}  {{/items}}  {{**^**items}}  Pas d'item  {{/items}}  </section>  <section>  {{**!** value contient '<b>Hello</b>'}}  {{value}}  {{**{**value**}**}}  </section> | | **>** => ajout de la page "partial-header.mustache" à la page  **#** => si la liste existe, alors pour chaque item  => item.name  **^** => si la liste items n'existe pas  **!** =>commentaire  =>caractères HTML échappées pour éviter l'injection (ex : &lt;b>Hellob&gt)  **{ }** =>caractère HTML non échapés (ex : <b>Hello</b>) |  |
|  |  |  | |  |  |
| Exceptions et pages d'erreurs |  |  | |  |  |
| templates/error | emplacement des pages d'erreurs personnalisées | src/main/resources/templates/error/ | |  |  |
| 404.mustache | page appelée automatiquement lors de la levée d'une erreur 404 | src/main/resources/templates/error/404.mustache | |  |  |
| application.properties | à ajouter dans **application.properties** pour permettre un affichage plus complet des messages d'erreurs dans les pages | server.error.include-stacktrace=always | |  |  |
| status  error  timestamp  path  message  trace | accès aux valeurs des champs de l'erreur dans un fichier .mustache | <h1>{{status}} : {{error}}</h1>  <ul>  <li>Heure : {{timestamp}}</li>  <li>Path : {{path}}</li>  <li>Message : {{message}}</li>  </ul>  <code>{{trace}}</code> | |  |  |
| @ControllerAdvice | Controller permettant de rediriger une exception vers une page d'erreur/fournir une erreur au client REST | @ControllerAdvice  public class ExceptionController {  *//Redirection vers une page d'erreur*  @ExceptionHandler  public ModelAndView handle(NotFoundException e){  ModelAndView mv = new ModelAndView("/error/404")  .addObject("status", 404)  .addObject("error", "Not found exception")  .addObject("trace", e.getStackTrace().toString())  .addObject("timestamp", new Date())  .addObject("message", e.getMessage());  mv.setStatus(HttpStatus.NOT\_FOUND);  return mv;  }  *//réponse d'erreur à la demande REST*  @ExceptionHandler(NotFoundException.class)  public ResponseEntity<ErrorDto> handle(NotFoundException e){  return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(new ErrorDto(e.getMessage()));  }  private class ErrorDto{  private final String message;  private final Date date = new Date();  public ErrorDto(String message) {  this.message = message;  }  public String getMessage() {  return message;  }  public Date getDate() {  return date;  }  }  } | | | Rq: par défaut, une exception est transformée en erreur 500 (= erreur du serveur) |
|  |  |  | |  |  |
|  |  |  | |  |  |
| REST | = REpresentational State Transfer |  | |  |  |
| API REST  ou  API Web | au lieu de renvoyer des données sous forme de pages web interprétable par le navigateur, on peut exposer les données de notre application via l'API REST incluse dans Spring.  Les données (toujours demandées via des URL) sont transmissent cette fois au format JSON pour être utilisé par une autre appli, par du JavaScript, ...  Pour cela, on ne passe plus par les Controller, mais par des RestController qui ne renverrons pas des ModelAndView, mais des objets DTO contenant les données qui seront automatiquement transcrites en JSON par Spring avant d'être renvoyées au client  **DTO** = Data Transfer Object (objet de transfert de données) | | | | |
| @RestController | définit un RestController qui renvoi des données au format JSON | @RestController  @RequestMapping("/sites/{siteId}/captors")  @Transactional  public class RestCaptorController {  @Autowired  private CaptorDao captorDao;  private List<CaptorDto> toDtos(List<Captor> captors){  return captors.stream().map(this::toDto).collect(Collectors.toList());  }  @GetMapping  public List<CaptorDto> findAll(@PathVariable String siteId) {  return toDtos(captorDao.findBySiteId(siteId));  }  @GetMapping  public List<CaptorDto> findAll(@RequestParam(required = false) String siteId, @RequestParam("monParametre") MonObjet monParaSousUnAutreNom) {  return toDtos(captorDao.findBySiteId(siteId));  }  @GetMapping("/{id}")  public CaptorDto findById(@PathVariable String siteId, @PathVariable String id) {  Captor captor = captorDao.findById(id).orElseThrow(NotFoundException::new);  return CaptorDto.toDto(captor);  }  @PostMapping  public CaptorDto save(@PathVariable String siteId, CaptorDto captorDto) {  Site site = siteDao.findById(siteId).orElseThrow(NotFoundException::new);  Captor captor = captorDto.toCaptor(site);  return CaptorDto.toDto(captorDao.save(captor));  }  @PutMapping("/{id}")  public void update(@PathVariable String siteId, @PathVariable String id, CaptorDto captorDto) {  Captor captorToUpdate = captorDao.findById(id).orElseThrow(NotFoundException::new);  captorToUpdate.setName(captorDto.getName());  captorDao.save(captorToUpdate);  }  @DeleteMapping("/{id}")  public void delete(@PathVariable String siteId, @PathVariable String id) {  measureDao.deleteByCaptorId(id);  captorDao.deleteById(id);  }  } | | |  |
| @Service | A l'inverse, si on veut utiliser l'API REST d'une autre application via une URL, il faut créer une classe Service qui va utiliser RestTemplate | @Service  public class MonServiceImpl implements MonService {  private final RestTemplate restTemplate;  public MonService(RestTemplateBuilder builder) {  *//interruption de la demande si délai supérieur à 1s*  this.restTemplate = builder.setConnectTimeout(1000).build(); }  @Override  public UserDto readMeasures(String userId) {  UriComponentsBuilder builder = UriComponentsBuilder  .fromUriString("http://urlappli/users/{user\_id}") *//url de l'appli REST et contenant la demande*  .build(userId);  *//appel du service en précisant le type d'objet qui servira pour récupérer les infos après lecture des données JSON :*  return this.restTemplate.getForObject(builder.toUriString(), UserDto.class);  }  } | | |  |
|  | Exemple avec une liste d'objet  et une url avec des paramètres : | @Service  public class SimulatedMeasureService implements MeasureService<SimulatedCaptor> {  private RestTemplate restTemplate;    public SimulatedMeasureService(RestTemplateBuilder builder) {  this.restTemplate = builder.setConnectTimeout(1000).build();  }  public List<Measure> readMeasures(SimulatedCaptor captor, Instant start, Instant end, MeasureStep step) {  checkReadMeasuresAgrs(captor, start, end, step);  UriComponentsBuilder builder = UriComponentsBuilder  .fromHttpUrl("http://localhost:8090/measures")  .path("")  .queryParam("start", start)  .queryParam("end", end)  .queryParam("min", captor.getMinPowerInWatt())  .queryParam("max", captor.getMaxPowerInWatt())  .queryParam("step", step.getDurationInSecondes());  Measure[] measures = this.restTemplate.getForObject(builder.toUriString(), Measure[].class);  return Arrays.asList(measures);  }  } | | |  |
|  |  |  | |  |  |
| Security |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |