

écosystème Spring

l'essentiel

Table des matières

I - Spring (Présentation, vue d'ensemble).....	11
1. Architecture / Ecosystème Spring.....	11
2. Design Pattern "I.O.C." / injection de dépendances.....	13
2.1. <i>IOC = inversion of control</i>	13
2.2. <i>injection de dépendance</i>	14
2.3. avec conteneur I.O.C. (<i>super fabrique globale</i>).....	15
2.4. <i>Micro-kernel / conteneur léger</i>	15
3. "Spring Container" et composants.....	16
4. Spring-Framework et SpringBoot.....	17
4.1. <i>SpringBoot en tant que sur-couche de spring-framework</i>	17
4.2. <i>Configuration globale d'une application spring-boot</i>	19
5. Principaux Modules de Spring-Framework.....	20

6. Configurations Spring – vue d'ensemble.....	21
6.1. Historique et évolution.....	21
6.2. Avantages et inconvénients de chaque mode de configuration.....	23
II - Configuration de Spring et tests essentiels.....	25
1. Vue d'ensemble sur config Spring.....	25
1.1. Complémentarité nécessaire / configuration mixte.....	25
1.2. Démarrages possibles depuis spring 4,5,6.....	26
1.3. Vue d'ensemble sur les aspects de la configuration spring.....	26
1.4. Configuration structurée (properties , import , profiles).....	27
2. Ancienne configuration Xml de Spring (aperçu).....	28
3. Configuration IOC Spring via des annotations.....	29
3.1. Annotations (stéréotypées) pour composant applicatif.....	29
3.2. Autres annotations ioc (@Required , @Autowired , @Qualifier).....	30
3.3. @Autowired (fondamental).....	30
3.4. Paramétrage des @ComponentScan de "Spring".....	32
3.5. Configuration minimum de démarrage (appli. Spring).....	33
3.6. @Qualifier (pour variantes qui peuvent coexister).....	34
3.7. rares @Scope et @Lazy.....	35
4. Cycle de vie , @PostConstruct , @PreDestroy.....	36
5. Injection par constructeur (assez conseillé).....	37
6. "Java Config" mieux qu'ancienne config xml.....	38
7. Java Config (Spring) en fonction du contexte.....	38
8. Config java élémentaire.....	39
8.1. Exemple1: DataSourceConfig :.....	39
8.2. Utilisations possibles (ici sans spring-boot):.....	40
8.3. Avec placeHolder et fichier ".properties".....	41
8.4. @Value avec Spring-EL.....	42
8.5. Quelques paramétrages (avancés) possibles :.....	42
8.6. injections de dépendances entre @Bean.....	43
8.7. Exemple concret: JpaExplicitConfig:.....	44
8.8. @Import explicites et implicites/automatiques.....	45
8.9. Profiles "spring" (variante de configuration).....	46
Un profil spring est une variante de configuration (avec nom libre et signification à définir).....	46
NB : certains profils peuvent être exclusifs (ex : "dev" ou bien "prod" , "withSecurity" ou bien "withoutSecurity") et d'autres peuvent être complémentaires (ex : "dev" et "withSecurity").....	46
Déclarations/définitions des variantes :.....	46
9. Tests "Junit4/5 + Spring" (spring-test).....	47

III - Spring Boot (l'essentiel).....	50
1. Fonctionnalités de SpringBoot.....	50
2. Spring-Initializer.....	52
3. Spring-boot (configuration et démarrage).....	54
3.1. Exemple de démarrage avec Spring-Boot.....	54
3.2. Configuration maven pour spring-boot 3 (et spring 6).....	55
3.3. Rare boot (standalone) sans annotation.....	56
3.4. Boot (standalone) avec annotation @SpringBootApplication.....	56
3.5. Structure minimaliste d'une application SpringBoot :.....	57
3.6. Tests unitaires avec Spring-boot.....	58
3.7. auto-configuration (facultative mais conseillée).....	59
3.8. Configuration des logs avec springBoot.....	59
3.9. Auto-configuration "spring-boot" avec application.properties.....	60
3.10. Profiles 'spring" (variantes dans les configurations).....	61
IV - Spring AOP (intercepteurs, aspects).....	63
1. Spring AOP (essentiel).....	63
1.1. Technologies AOP et "Spring AOP".....	63
1.2. Mise en oeuvre rapide de Spring aop via des annotations.....	64
1.3. Types d'advice.....	65
1.4. Spring-AOP à l'ancienne (avec Advisor/intercepteur , proxies).....	68
V - Accès aux données depuis Spring (jdbc,JPA).....	71
1. Accès au données via Spring (possibilités).....	71
2. Utilisation de Spring au niveau des services métiers.....	71
2.1. Dépendances classiques.....	71
2.2. Principales fonctionnalités d'un service métier.....	72
2.3. Vision abstraite d'un service métier.....	72
3. DataSource JDBC (vue Spring).....	73
3.1. DataSource élémentaire (sans pool).....	73
3.2. Embedded DataSource with pool.....	74
4. DAO Spring basé directement sur JDBC.....	75
4.1. Avec JdbcDaoSupport.....	75
4.2. Avec JdbcTemplate.....	77
4.3. Avec NamedParameterJdbcTemplate et RowMapper.....	79
5. Intégration de JPA/Hibernate dans Spring.....	81
6. DAO Spring basé sur JPA (Java Persistence Api).....	81
6.1. Rappel: Entité prise en charge par JPA.....	81
6.2. unité de persistance (persistence.xml facultatif).....	81
6.3. Configuration "spring / jpa" classique (en version xml)	82
6.4. TxManager compatible JPA et @PersistenceContext.....	82

6.5. Configuration Jpa / Spring (sans xml) en mode java-config.....	83
6.6. Simplification "Spring-boot" et @EnableAutoConfiguration.....	84
6.7. DAO «JPA» style «pure JPA,Ejb3» pris en charge par Spring.....	86
VI - Spring-Data (l'essentiel).....	87
1. Spring-Data.....	87
1.1. Spring-data-commons.....	87
1.2. Spring-data-jpa.....	89
1.3. Autres parties existantes de Spring-data.....	94
1.4. Essentiel sur Spring-data-mongo.....	94
VII - Transactions Spring.....	96
1. Support des transactions au niveau de Spring.....	96
2. Propagation du contexte transactionnel et effets.....	98
3. Configuration du gestionnaire de transactions.....	99
3.1. Différentes implémentations de PlatformTransactionManager.....	99
3.2. Exemple de configuration explicite en mode "java-config".....	99
4. Marquer besoin en transaction avec @Transactional.....	100
VIII - Architectures web / spring (vue d'ensemble).....	101
1. Spring web overview.....	101
2. Vue d'ensemble sur Spring asynchrone.....	103
IX - Spring-web (avec serveur, génération HTML).....	105
1. Lien avec un serveur JEE et Spring web.....	105
2. Injection de Spring au sein d'un framework WEB.....	105
2.1. WebApplicationContext (configuration xml).....	105
2.2. WebApplicationContext (configuration java-config).....	105
2.3. WebApplicationContext (accès et utilisation).....	106
3. Packaging d'une application SpringBoot pour un déploiement vers un serveur JEE.....	107
4. Présentation du framework "Spring MVC".....	108
4.1. configuration maven pour spring-mvc.....	108
5. Mécanismes fondamentaux de "spring mvc".....	109
5.1. Principe de fonctionnement de SpringMvc :.....	109
5.2. Exemple Spring-Mvc simple en version ".jsp":.....	110
X - Api REST via spring-Mvc et OpenApiDoc.....	111
1. Web services "REST" pour application Spring.....	111
2. Variantes d'architectures.....	112

3. WS REST via Spring MVC et @RestController.....	115
3.1. Gestion des requêtes en lecture (mode GET).....	115
3.2. @RequestBody et ResponseEntity<T>.....	116
3.3. Variantes de code :	119
Exemple de code pour "POST" et "PUT" :	120
//appelé en mode POST.....	120
//avec url = http://localhost:8181/appliSpring/rest/api-xyz/v1/xyz.....	120
//avec dans la partie "body" de la requête { "id" : null , "label" : "..." , "price" : 50.0 }.....	120
@PostMapping("").....	120
public ResponseEntity<?> postXyz(@Valid @RequestBody XyzToCreate obj) {.....	120
Xyz savedObj = serviceXyz.create(obj); //avec id auto_incrémenté.....	120
.....	120
URI location = ServletUriComponentsBuilder.....	120
.fromCurrentRequest().....	120
.path("/{id}").....	120
.buildAndExpand(savedObj.getId()).toUri().....	120
//return ResponseEntity.created(location).build().....	120
//return 201/CREATED , no body but URI to find added obj.....	120
return ResponseEntity.created(location).body(savedObj);.....	120
//return 201/CREATED with savedObj AND with URI to find added obj.....	120
/* ou bien encore return ResponseEntity.ok().....	120
.headers(responseHeadersWithLocation).body(savedObj);.....	120
*/.....	120
}.....	120
Résultat :	120
//à appeler en mode PUT.....	120
//avec url = http://localhost:8181/appliSpring/rest/api-xyz/v1/xyz/1.....	120
//avec dans la partie "body" de la requête { "id" : 1 , "label" : "..." , "price" : 120.0 }.....	120
@PutMapping("/{id}").....	120
public ResponseEntity<Xyz> putCompte(@RequestBody Xyz obj, @PathVariable("id") Long idToUpdate) {.....	120
obj.setId(idToUpdate);.....	120
Xyz updatedObj = serviceXyz.update(obj);.....	120
return ResponseEntity.status(HttpStatus.NO_CONTENT).build();.....	120
//204 : OK sans aucun message dans partie body.....	120
//exception handler may return NOT_FOUND or INTERNAL_SERVER_ERROR.....	120
}.....	120
Résultat :	120
3.4. Réponse et statut http par défaut en cas d'exception.....	121
3.5. @ResponseStatus.....	121
3.6. ResponseEntityExceptionHandler (très bien).....	122
3.7. Validation des valeurs entrantes (@Valid).....	124
Résultat en cas de données en entrée incorrectes:	124
3.8. Exemples d'appels en js/ajax.....	125

3.9. Différentes API pour appeler des WS REST via Spring.....	130
3.10. Invocation java de service REST via RestTemplate de Spring.....	130
3.11. Appel moderne/asynchrone de WS-REST avec WebClient.....	134
3.12. Invocation via l'api standard HTTP2 de java ≥ 9.....	136
3.13. Via RestClient de Spring ≥ 6.1.....	139
3.14. Test d'un "RestController" via MockMvc.....	142
3.15. Test unitaire de contrôleur Rest.....	142
3.16. Test d'intégration de contrôleur Rest avec réels services.....	145
4. Config swagger3 / openapi-doc pour spring.....	146

XI - Spring Security (l'essentiel)..... 156

1. Extension Spring-security (généralités).....	156
1.1. Principales fonctionnalités de spring-security.....	156
1.2. Principaux besoins types (spring-security).....	157
1.3. Filtre web et SecurityFilterChain.....	160
1.4. Multiple SecurityFilterChain.....	161
1.5. Vue d'ensemble sur les phases de Spring-security.....	163
1.6. Comportement de l'authentification (spring-security).....	163
1.7. Mécanismes d'authentification (spring-security).....	164
1.8. Vue d'ensemble sur configuration concrète de la sécurité.....	165
1.9. Encodage classique des mots de passe via BCrypt.....	166
1.10. Prise en compte d'une authentification vérifiée.....	167
2. Configuration des "Realms" (spring-security).....	168
2.1. Principales implémentations possibles des realms.....	168
2.2. Utilisation classique d'un realm.....	169
2.3. AuthenticationManagerBuilder.....	170
2.4. Délégation d'authentification (OAuth2/Oidc).....	171
2.5. Realm temporaire "InMemory".....	171
2.6. Authentification jdbc ("realm" en base de données).....	171
2.7. Authentification "personnalisée" en implémentant l'interface UserDetailsService ..	174
3. Configuration des zones(url) à protéger.....	176
3.1. Généralités sur la configuration de HttpSecurity.....	176
3.2. Configuration type pour un projet de type Thymeleaf ou JSP.....	177
3.3. Champ caché "_csrf" de spring-mvc utile pour pages/vues "java/jsp" mais inutile pour Api-REST avec tokens	178
3.4. Configuration type pour un projet de type "Api REST".....	179
3.5. Paramétrage des autorisations selon rôles ou scopes.....	180
3.6. Eventuelle configuration d'autorisations "CORS".....	180

XII - Tests avancés avec Spring..... 181

1. Différents types de tests (environnement spring).....	181
2. Test purement unitaire.....	181

3. Test spring avec DirtyContext.....	184
4. Test d'intégration partiel.....	186
4.1. Version 1 sans @MockBean.....	186
4.2. Version 2 avec @MockBean.....	188
5. Optimisation sur la partie à charger et tester.....	189
5.1. Via configuration adéquate.....	189
5.2. @DataJpaTest.....	189
6. Test "end-to-end" sur backend.....	189

XIII - Annexe – Selon versions (ajustements).....191

1. Migration de Spring 5 vers Spring 6.....	191
1.1. Changement de dépendances (pom.xml).....	191
1.2. Changement de packages.....	192
1.3. Continuité vis à vis des évolutions de SpringSecurity.....	192
1.4. Changements dans les paramétrages de l'auto-configuration.....	193
1.5. Nouvelles possibilités de java 17.....	193
1.6. Nouvelles possibilités de Spring 6.....	193

XIV - Annexe – Configuration Spring avancée.....194

1. "java config" avancée.....	194
1.1. Démarrage souple/contextuel.....	194
1.2. Chargement automatique d'un paquet de propriétés dans un objet java (avec éventuels sous objets).....	196
1.3. @Primary (version principale/prioritaire).....	197
1.4. "dataSource" secondaire (multiple).....	198
1.5. ApplicationContextAware.....	199
2. Configuration avancée pour SpringBoot.....	200
2.1. Configuration conditionnelle intelligente.....	200
2.2. auto-configuration.....	201

XV - Annexe – Spring-MVC (JSP et Thymeleaf).....202

1. Spring-MVC avec pages JSP.....	202
1.1. Dépendances maven (SpringMvc + JSP).....	202
1.2. Configuration en version ".jsp".....	202
1.3. Exemple élémentaire (SpringMvc + JSP):.....	203
2. éléments essentiels de Spring web MVC.....	204
2.1. éventuelle génération directe de la réponse HTTP.....	204
2.2. @RequestParam (accès aux paramètres HTTP).....	204
2.3. @ModelAttribute.....	205
2.4. @SessionAttributes.....	207
2.5. tags pour formulaires JSP (form:form , form:input , ...).....	207
2.6. validation lors de la soumission d'un formulaire.....	210

3. Spring-Mvc avec Thymeleaf.....	212
3.1. Vues en version Thymeleaf.....	212
3.2. Spring-mvc avec Thymeleaf.....	212
3.3. "Hello world" avec Spring-Mvc et Thymeleaf.....	214
3.4. Templates thymeleaf avec layout.....	215
3.5. Principales syntaxes pour thymeleaf.....	220
3.6. Sécurité avec SpringMvc et thymeleaf.....	221
XVI - Annexe – async (reactor , webflux).....	223
1. ReactiveStreams" avec Reactor.....	223
2. WebFlux.....	225
2.1. Reactive WebClient.....	225
2.2. Reactive WebController.....	225
XVII - Annexe – Spring Actuator.....	226
1. Spring-Actuator.....	226
1.1. Présentation de "spring-actuator".....	226
1.2. Mise en oeuvre de spring-actuator.....	226
1.3. URLs des "endpoints" de spring actuator :	227
1.4. Paramétrages (application.properties) de actuator/info.....	230
1.5. Codage d'un "health indicator" spécifique.....	231
1.6. SpringBootAdmin (extension de de.codecentric).....	233
XVIII - Annexe – Web Services REST (concepts).....	238
1. Deux grands types de WS (REST et SOAP).....	238
1.1. Caractéristiques clefs des web-services "REST" / "HTTP".....	239
2. Web Services "R.E.S.T."	240
2.1. Statuts HTTP (code d'erreur ou ...).....	244
2.2. Variantes classiques.....	245
2.3. Safe and idempotent REST API.....	246
3. Test de W.S. REST via Postman.....	247
3.1. paramétrages "postman" pour une requête en mode "post".....	247
3.2. Exemple de réponses précises reçues et affichées par "postman".....	248
4. Test de W.S. REST via curl.....	249
5. Api Key.....	251
6. Token d'authentification.....	252
6.1. Tokens : notions et principes.....	252
6.2. Bearer Token (au porteur) / normalisé HTTP.....	254
6.3. JWT (Json Web Token).....	255
XIX - Annexe – WebSockets , STOMP (Spring).....	256

1. WebSockets HTML5 (standard).....	256
1.1. Fonctionnalités des WebSockets.....	256
1.2. Principe de fonctionnement.....	256
1.3. Exemple de "chat"(webSocket) : code client "html5+javascript".....	257
2. WebSockets avec Spring et STOMP	259
2.1. STOMP.....	259
2.2. Configuration.....	259

XX - Annexe – Spring Native Image.....264

XXI - GraalVM et Graal Native image.....	264
1.1. GraalVM.....	264
1.2. GraalVM Native Image.....	264
1.3. Spring Native.....	265
1.4. Quarkus (de RedHat).....	265
XXII - Spring Native Image.....	266
1.1. Petit exemple (native-image).....	267

XXIII - Annexe – énoncés des Tps / Spring.....269

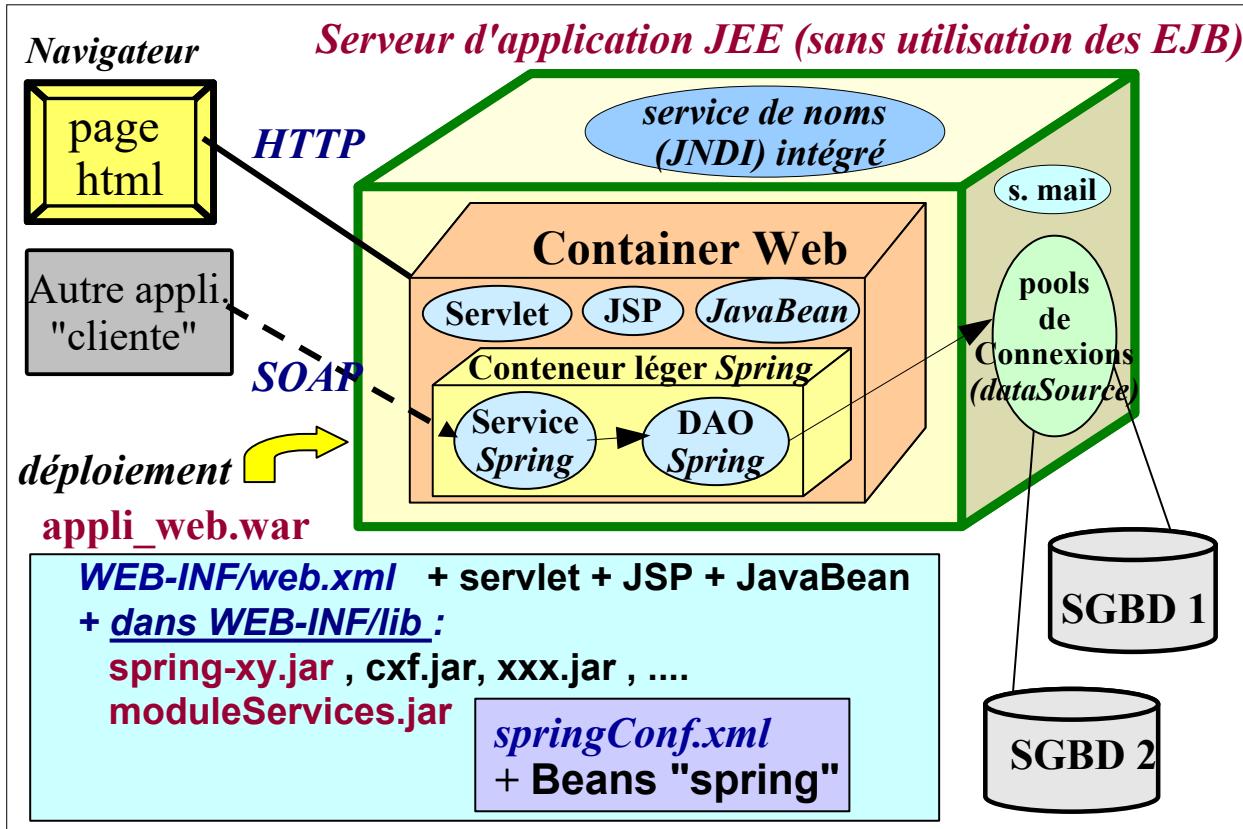
1. Tp sur bases de spring-framework.....	269
1.1. (Tp facultatif) Très rapide aperçu sur ancienne config XML.....	269
1.2. Chargement du projet et analyses/vérifications.....	269
1.3. Bases de l'injection de dépendance (avec @Autowired).....	269
1.4. Configurations via classes java (@Configuration, @Bean).....	270
1.5. Mise en place d'un aspect de type "log automatique".....	271
2. Tp sur transaction spring et accès aux données (jdbc,jpa) sans spring-data.....	272
2.1. Accès aux données (DataSource JDBC) , DAO.....	272
2.2. Petit exemple de DAO via JDBCTemplate.....	276
2.3. Accès aux données via JPA/Hibernate.....	278
2.4. Service Spring et gestion des transactions.....	281
3. Tp sur transactions , spring-data (avec spring-boot).....	285
3.1. (Tp facultatif) , nouveau projet via spring initializr.....	285
3.2. Chargement du projet springBoot et analyses/vérifications.....	285
3.3. Analyse de application.properties et variantes (profiles).....	286
3.4. Dao en mode JpaRepository.....	286
3.5. Transactions sur virement bancaire.....	287
4. Tp sur Api-REST avec Spring-Mvc.....	290
4.1. Familiarisation avec la structure du projet.....	290
4.2. Partie "get" de l'api REST.....	290
4.3. Gestion des statuts Http et des exceptions.....	290
4.4. Partie "post,put,delete" de l'api REST.....	291
4.5. Validation des entrées avec @Valid.....	291

4.6. Test unitaire pour Api REST (<i>tp facultatif</i>).....	291
4.7. Utilisation de <i>mapStruct</i> (<i>tp facultatif</i>).....	291
4.8. Exemple d'appel REST externe (<i>tp facultatif</i>).....	292
5. Tp sur DHTML via SpringMvc et Thymeleaf (ou JSP).....	293
5.1. Analyse des exemples basiques et des templates <i>thymeleaf</i>	293
5.2. Tp @SessionAttribute sur calcul de racine carrée.....	293
5.3. Analyse d'autres exemples et expérimentations.....	293
5.4. TP Virement bancaire via SpringMVC et Thymeleaf.....	294
6. Tp sur Spring-security.....	295
6.1. A savoir avec <i>spring-boot-starter-security</i>	295
6.2. <i>withoutSecurity</i>	295
6.3. Sécurité sur Api REST en mode OAuth2/OIDC.....	295
6.4. Sécurité sur Api REST sans OAuth2 avec gestion directe JWT.....	296
6.5. Paramétrage @PreAuthorize sur CompteRestCtrl.....	296
6.6. Sécurité sur partie site (<i>thymeleaf</i>) / Tp facultatif.....	297
6.7. Test unitaire partiel facultatif de CompteRestCtrl en mode sécurisé.....	297
6.8. (facultatif) avec projet annexe "mysecurity-autoconfigure"	297
7. Tp sur packaging et supervision d'appli spring.....	298
7.1. (<i>tp facultatif</i>) Déploiement d'une application Spring6/SpringBoot3 au format ".war" vers tomcat10.....	298
7.2. (<i>tp facultatif</i>) Déploiement d'une application Spring6/SpringBoot4 via docker.....	298
7.3. (<i>tp facultatif</i>) actuator.....	298

I - Spring (Présentation, vue d'ensemble)

1. Architecture / Ecosystème Spring

Durant la première décennie du XXI siècle, Spring était essentiellement à considérer comme une alternative aux EJB et respectant les spécifications JEE :



Dès les premières versions, le framework open source "Spring" apportait les principales fonctionnalités suivantes :

- intégration de composants complémentaires inter-dépendants via le design-pattern "**injection de dépendances / ioc**". configuration souple et flexible
- prise en charge automatique et "déclarative" (via config xml ou annotations) des **transactions** (commit/rollback)
- intégration des principaux autres frameworks java/JEE (**Hibernate/Jpa** , **Struts** , **JSF** , **JDBC** , ...)
- intercepteurs (aop)
- tests unitaires simples (JUnit + spring-test)
- quelques éléments de sécurité (sécurité JEE simplifiée)
-

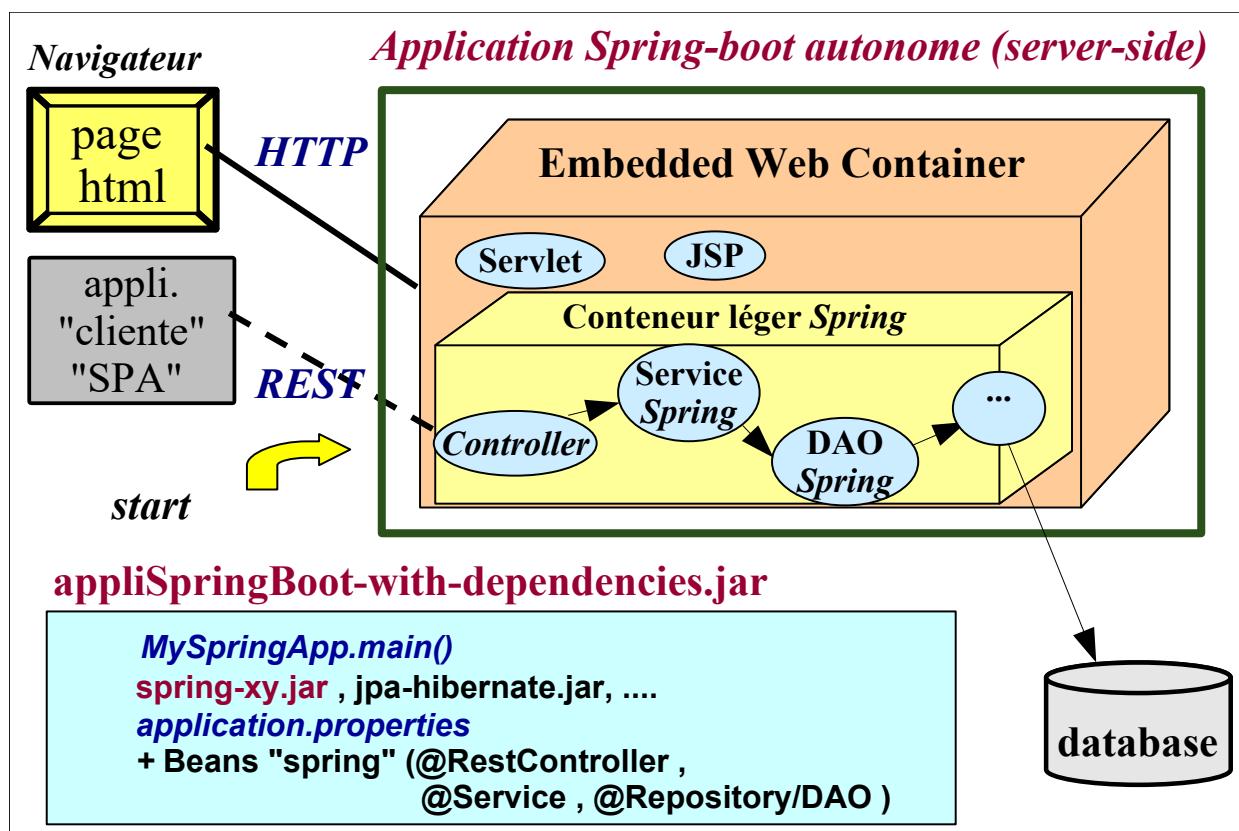
Le framework Spring n'est pas associé à un grand éditeur de serveur JEE (tel que IBM, Oracle/BEA, Jboss). Il a toujours laissé place à une très **grande liberté** dans le choix des technologies utilisées au sein d'une application Java/JEE.

A partir de la version 4, Le framework spring a introduit tout un tas de spécificités très intéressantes qui se démarquent clairement des spécifications JEE officielles .

Principales fonctionnalités supplémentaires apportées par les versions 4 ,5 et 6 de Spring :

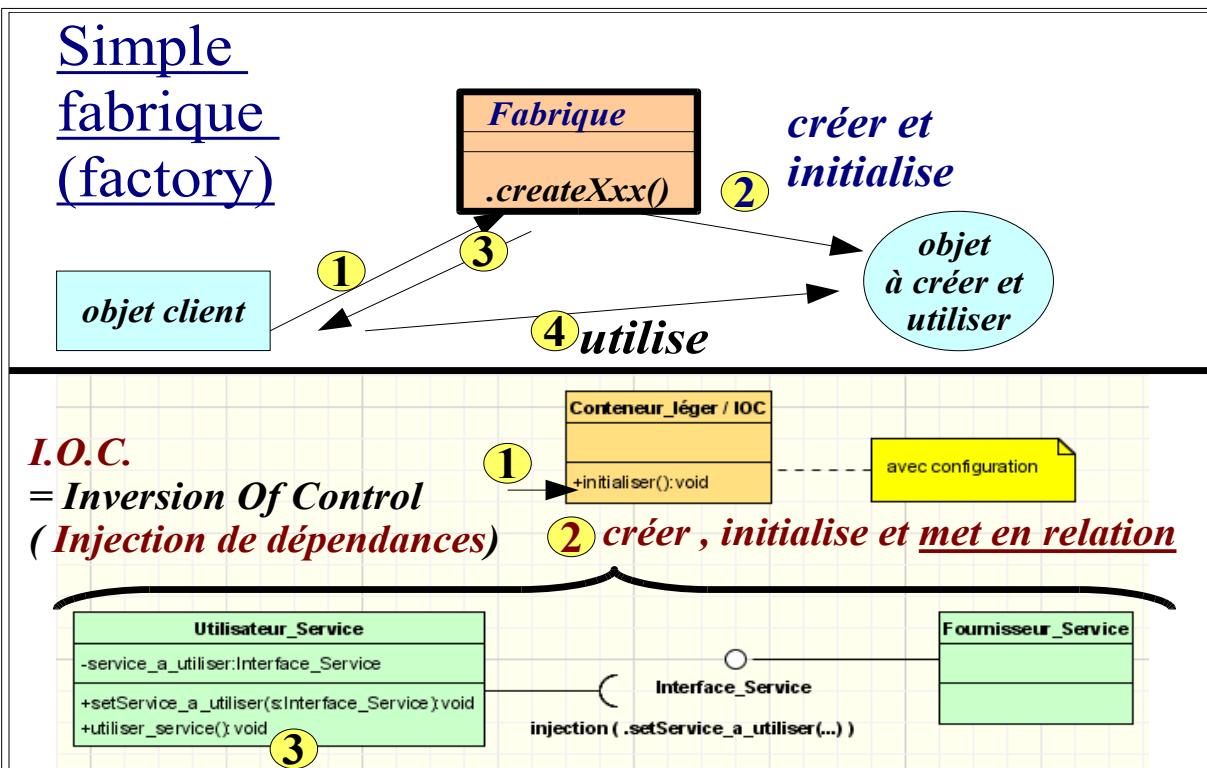
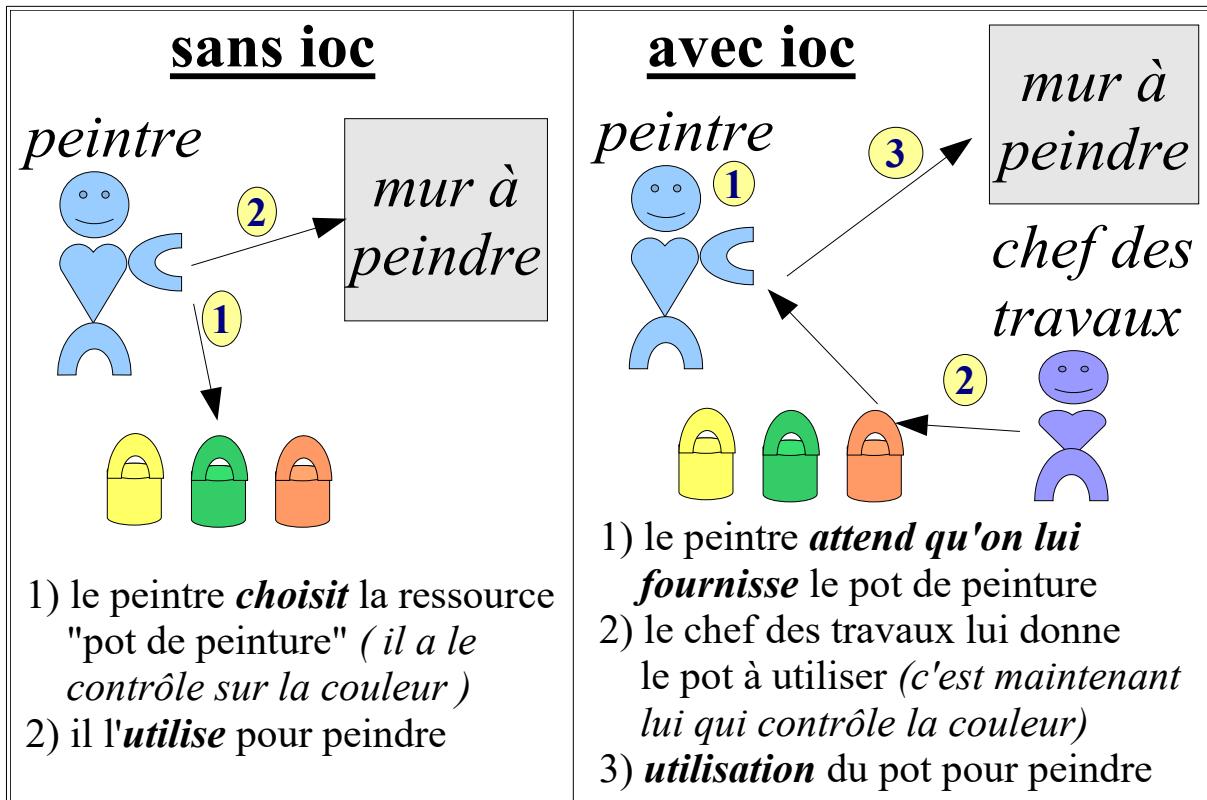
- **Spring boot** (démarrage complètement autonome . l'application incorpore son propre conteneur web (tomcat ou jetty ou netty ou ...))
- simplification de la configuration maven (ou gradle) via héritage de "POM/BOM/parent" .
- **Configuration java** (plus sophistiquée que l'ancienne configuration Xml , auto-complétion, rigueur , héritage , configuration conditionnelle intelligente)
- **AutoConfiguration** et simple fichier **application.properties** ou **.yml**
- **Spring Data** (composants "DAO" générés automatiquement à partir des signatures des méthodes d'une interface, implémentation possible via JPA et MongoDB , paramétrages possibles via @NamedQuery ou autres, ...)
- web services REST via **@RestController** de Spring-mvc
- sécurisation flexible via **Spring-security**
- autres fonctionnalités diverses (*actuators* : mesures de perf , ...) ,

Toutes ces fonctionnalités (bien pratiques) sont "hors spécifications JEE" et l'on peut aujourd'hui considérer que "**Spring**" forme un "**écosystème complet**" pour faire fonctionner des applications professionnelles "java/web".

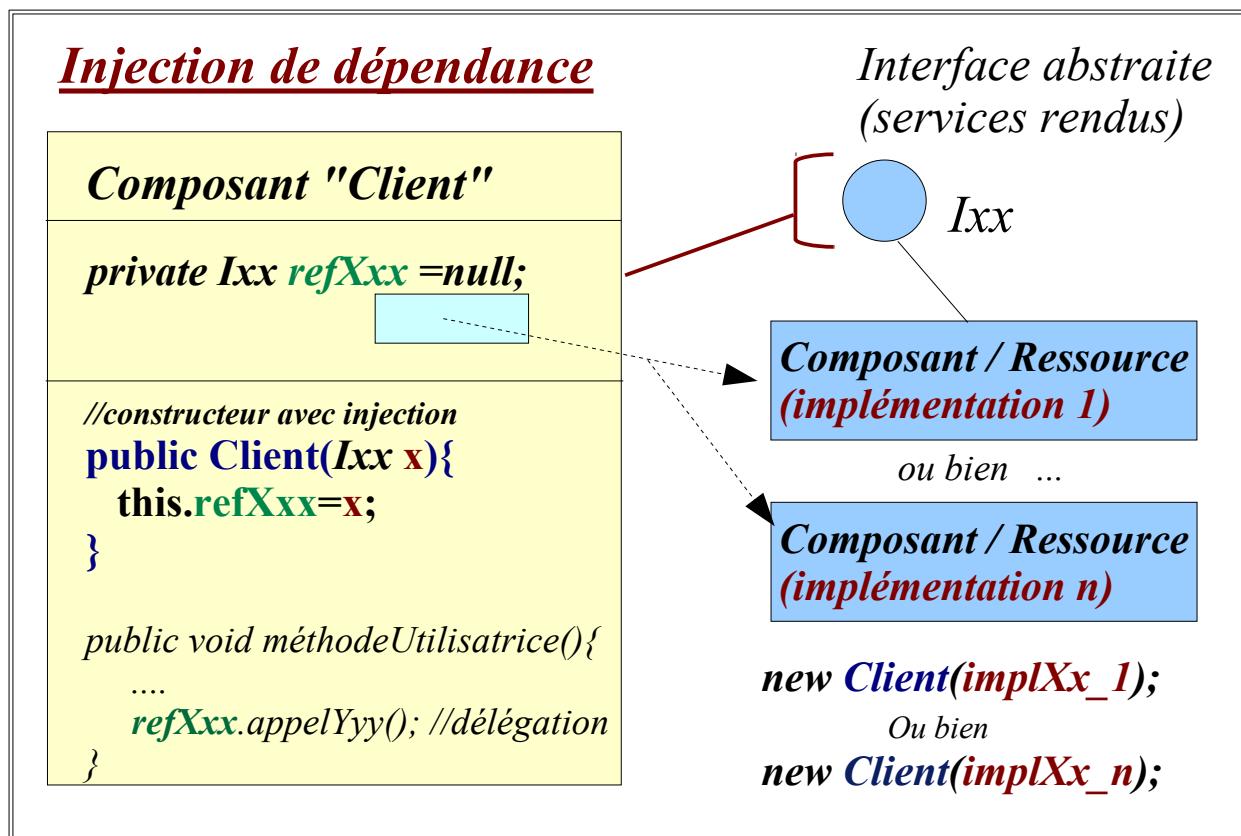


2. Design Pattern "I.O.C." / injection de dépendances

2.1. IOC = inversion of control



2.2. injection de dépendance



Le **design pattern "IOC"** (*Inversion of control*) correspond à la notion d'injection de dépendances abstraites.

Concrètement au lieu qu'un composant "client" trouve (ou choisisse) lui même une ressource compatible avec l'interface Ixx avant de l'utiliser , cet **objet client exposera une méthode de type:**

public void setRefXxx(Ixx res)

ou bien un constructeur de type:

public Client(Ixx res)

ou bien une référence annotée de type :

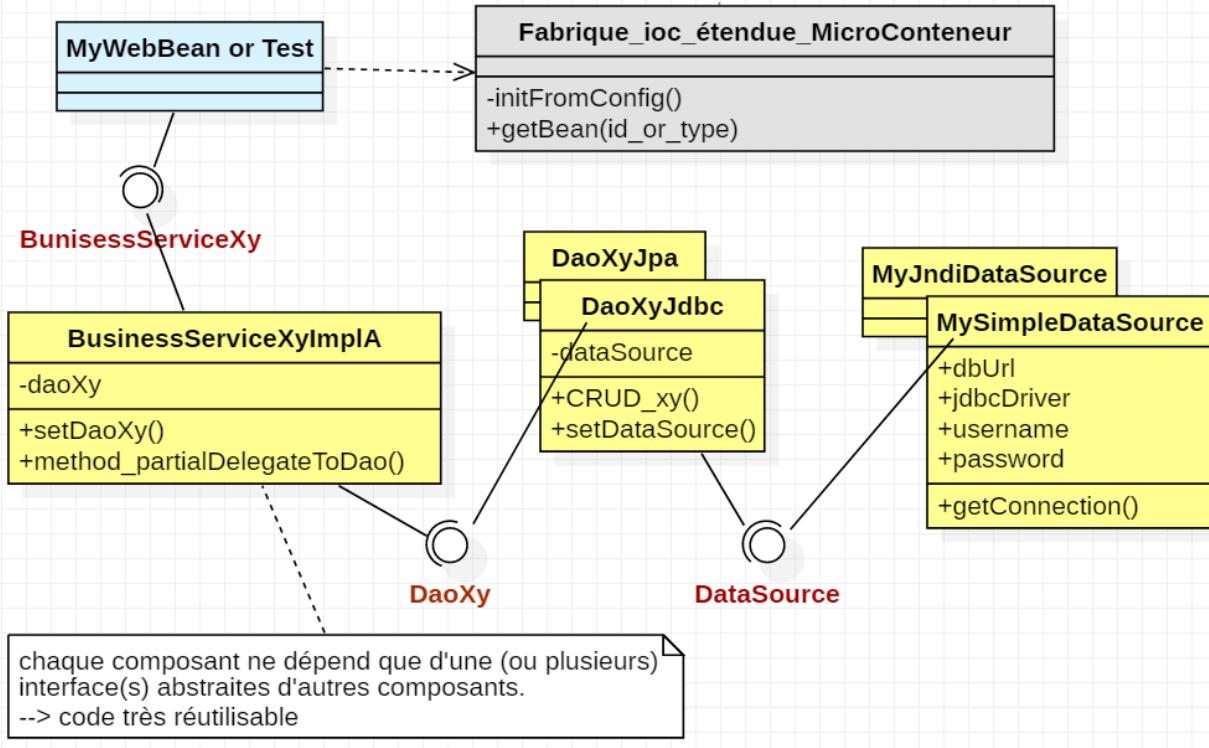
@Autowired ou bien **@Inject**

private Ixx refXxx;

permettant qu'on lui fournisse la ressource à ultérieurement utiliser. Un tel composant sera ainsi très réutilisable .

2.3. avec conteneur I.O.C. (super fabrique globale)

la méthode .getBean() va (à peu près) en interne:
 - instancier si besoin tous les éléments de la chaîne de composants
 - les relier entre eux en appelant automatiquement les méthodes d'injection de dépendance (bs.setDaoXy(daoXyJdbc); daoXyJdbc.setDataSource(ds1);
 - retourner le premier élément demandé de la chaîne construite et initialisée.



2.4. Micro-kernel / conteneur léger

Pour être facilement exploitable, le design pattern "injection de dépendances" nécessite un **petit framework** généralement appelé "**micro-kernel**" ou "**conteneur léger**" prenant à sa charge les fonctionnalités suivantes:

- **Enregistrement des "ressources"** (composants concrets basés sur interfaces abstraites) avec des **identifiants** (*noms logiques*) associés.
- **Instanciation et/ou initialisation des composants en tenant compte des dépendances à injecter** (==> liaisons automatiques avec composants "ressources" nécessaires)

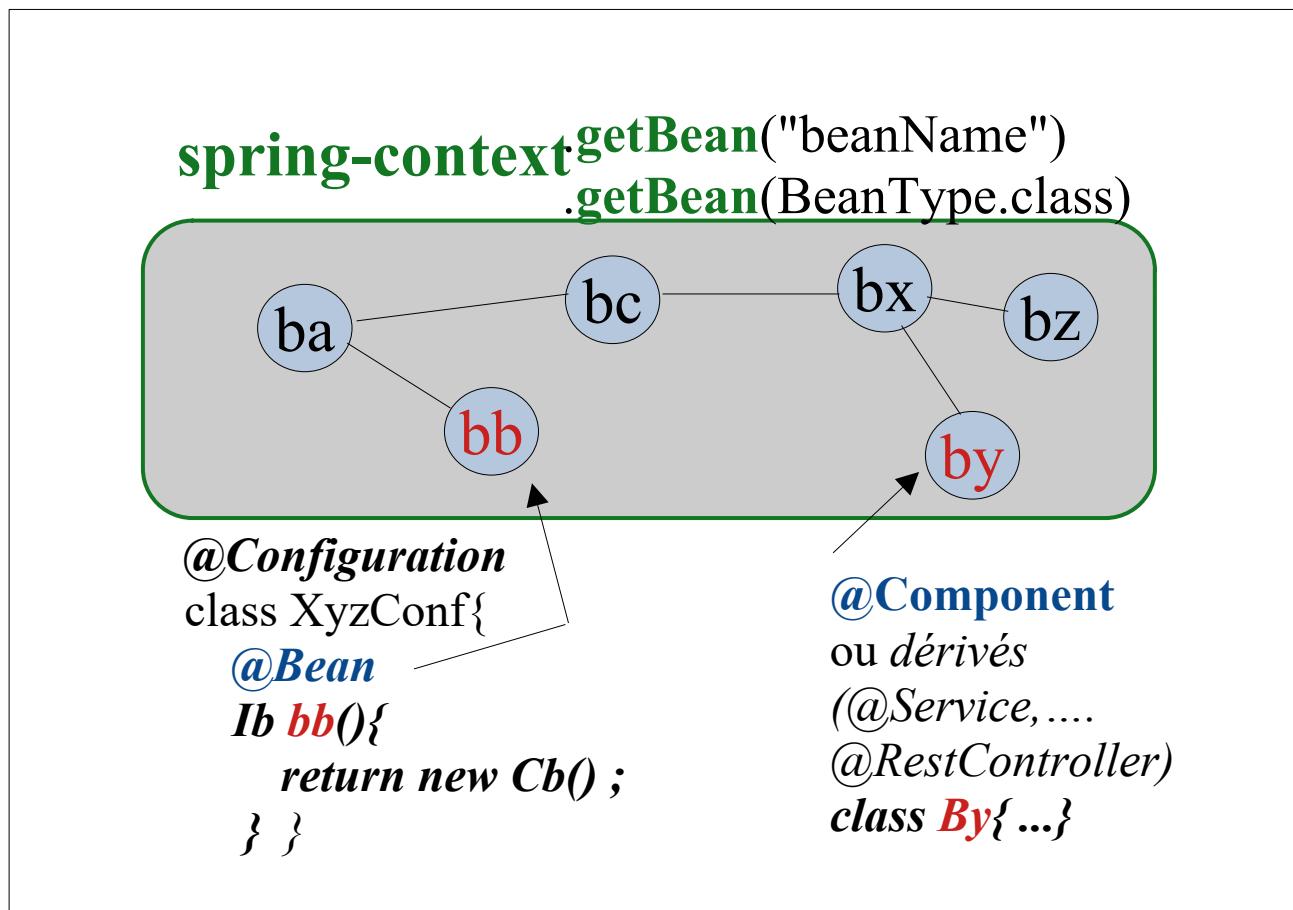
Ceci nécessite **quelques paramétrages** (*fichier de configuration XML* ou bien **annotations** au sein du code ou bien via une *configuration spécifique (java, implicite ou explicite, ...)*).

3. "Spring Container" et composants

Basé à fond sur le principe d'injection de dépendances , une application Spring est avant tout constituée d'un ensemble de composants gérés par le framework spring.

Notions fondamentales :

"Spring Container"	Le coeur de spring (socle technique pré-programmé prenant en charge les composants)
"spring context"	L'ensemble des objets java (composants) pris en charges par spring
Composant Spring (alias "bean")	Instance d'une classe java prise en charge par le conteneur spring. Chaque composant a un id (ou nom) unique (sous forme de String)

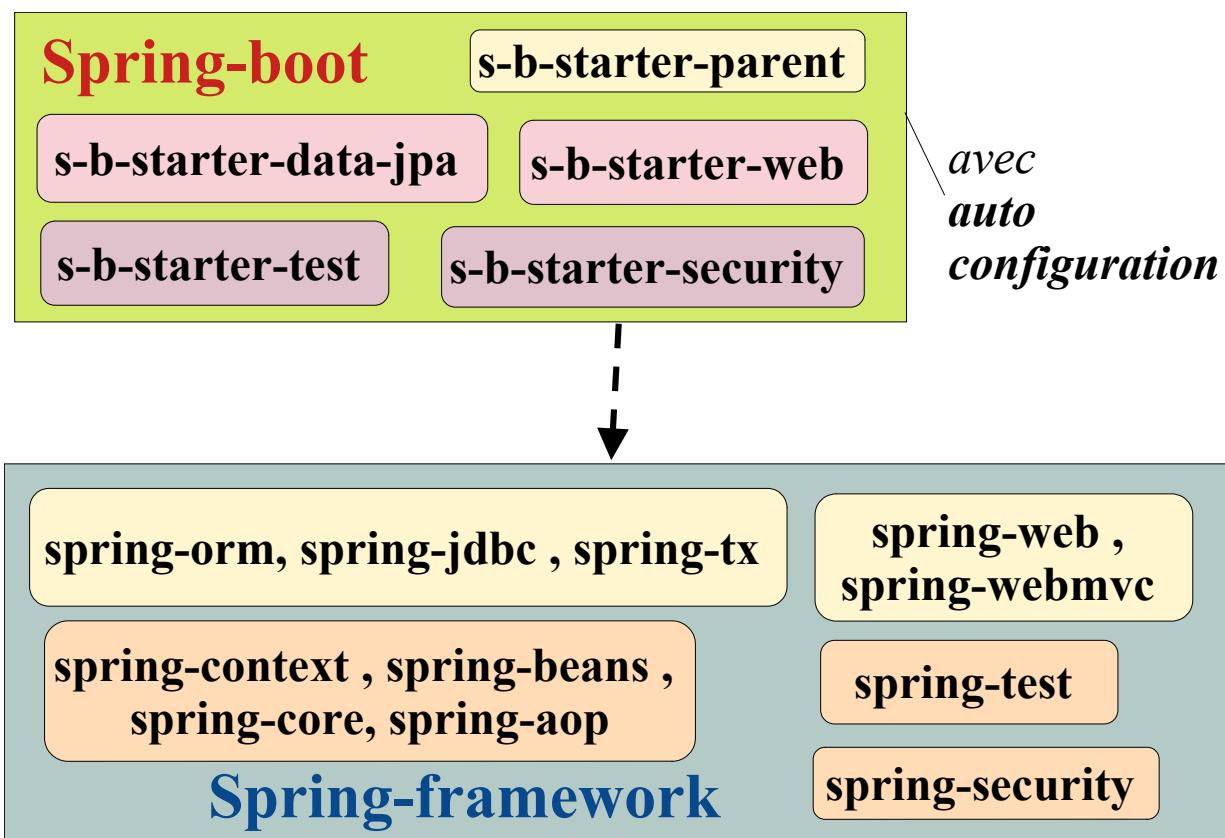


NB :

- le "spring_context" est créé et initialisé dès le démarrage de l'application .
- Ce contexte est configuré avec "classes java + annotations + fichier ".properties" ou ".yml"
- La méthode fondamentale `.getBean("id_name_of_beans")` ou `.getBean(TypeBean.class)` du contexte permet d'accéder à un des composants pris en charge par Spring
- Certains des composants sont configurés simplement comme des classes java comportant des annotations simples (**@Component**, **@Autowired**, ...)
- D'autres composants spring sont issus d'une configuration plus explicite (anciennement en xml , maintenant classe java de **@Configuration** avec méthode de fabrication de **@Bean**).

4. Spring-Framework et SpringBoot

4.1. SpringBoot en tant que sur-couche de spring-framework

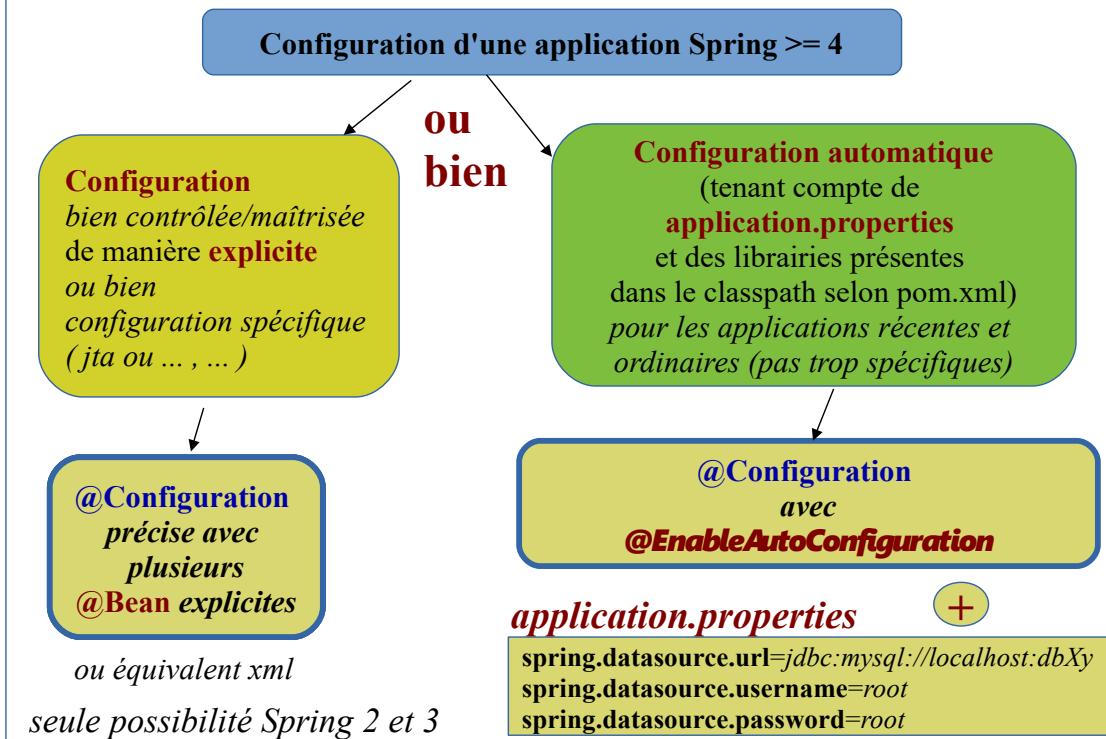


L'utilisation de SpringBoot reste facultative (bien que fortement conseillée dans la plupart des cas).

- **Spring-framework** est un **ensemble de librairies/modules** qu'il faut utiliser avec un minimum de configuration explicite
- **Spring-boot** est une **surcouche qui apporte essentiellement une configuration automatique** pour les besoins les plus courants

Application uniquement basée sur spring-framework (seule possibilité avant spring 4)	<i>Configuration explicite</i> (anciennement xml, maintenant en java) nécessaire pour configurer l'accès aux bases de données, la gestion des transactions et certains paramètres web (url, ...)
Application basée sur springBoot (possible depuis spring 4)	Juste besoin d'une <i>configuration allégée</i> (dans application.properties ou .yml)

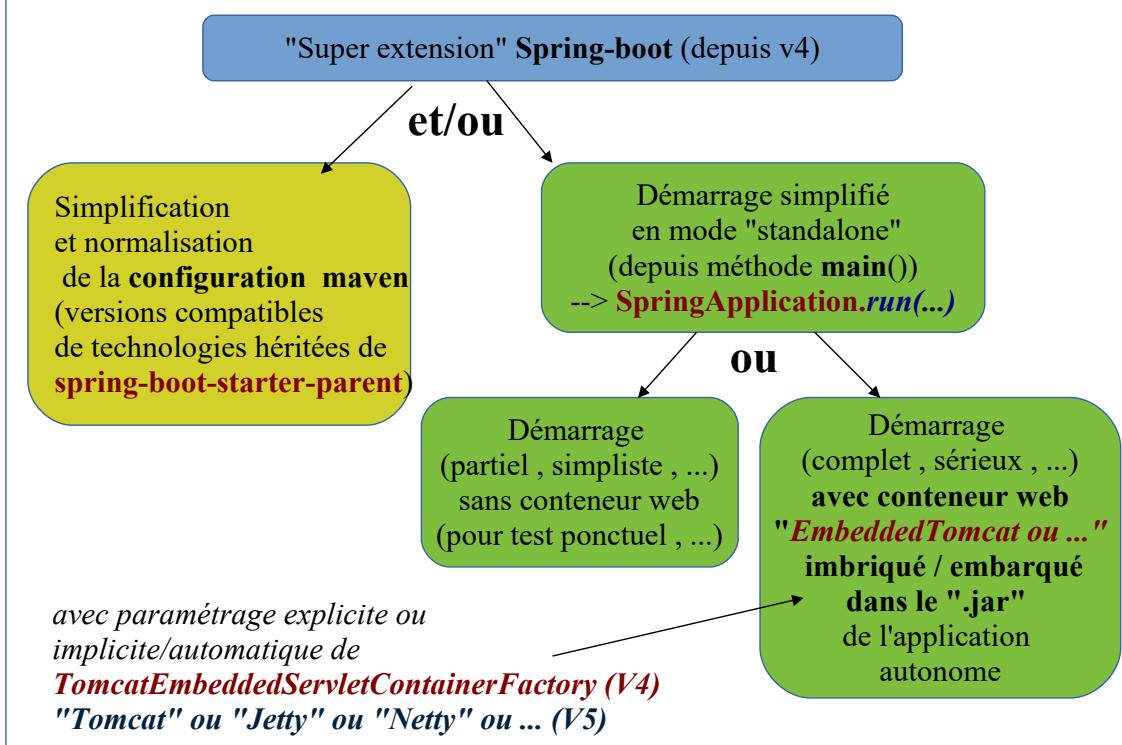
Spring >= 4 (éventuelle auto-configuration)



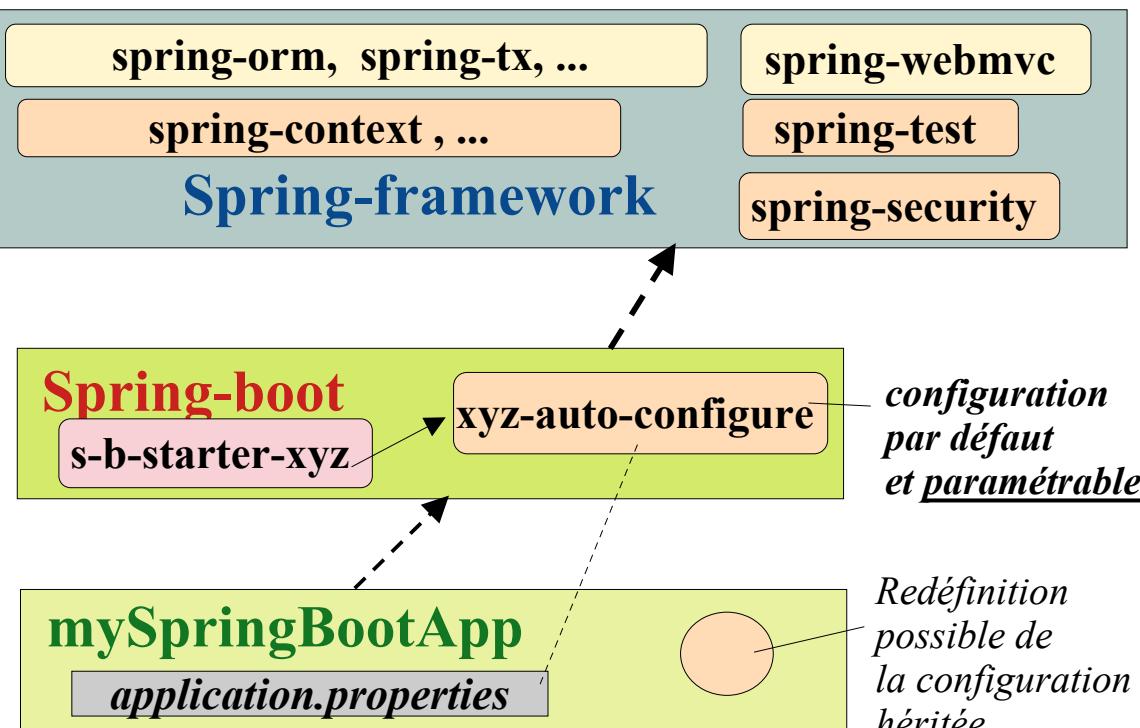
Autres apports de SpringBoot :

- simplification de la configuration maven (versions selon parent)
- démarrage possible en mode autonome (.jar/ main()) sans serveur

Spring >= 4 (apports facultatifs de Spring-boot)



4.2. Configuration globale d'une application spring-boot



- Une application **springBoot** moderne s'appuie sur un ensemble d'**auto-configurations de bas niveaux** (codées en java et cachées dans des librairies **xyz-auto-configure** elles-mêmes référencées par des **spring-boot-starter-xyz**)
- Ces **auto-configurations** vont automatiquement analyser des parties du fichier **application.properties** de manière à configurer certains aspects (url, accès aux base de données, ...)
- On a souvent besoin d'un minimum de configuration supplémentaire explicite (ex : sécurité)
- Une auto-configuration est une configuration par défaut (pas prioritaire) qui peut être redéfinie au cas par cas au sein d'une application ayant des besoins spécifiques .

NB : Certaines annexes ("java-config avancée", "config springBoot avancée") montrent comment se créer soit-même de nouveaux modules d'auto-configuration réutilisables .

5. Principaux Modules de Spring-Framework

Modules de Spring	Contenus / spécificités
Spring Core + Spring Beans	conteneur léger – IOC (base du framework – BeanFactory)
Spring AOP	prise en charge de la programmation orientée aspect
Spring DAO	Classes d'exceptions pour DAO (Data Access Object), Classes abstraites facilitant l'implémentation d'un DAO basé sur Hibernate ou JDBC. Infrastructure/support pour les transactions
Spring Context	Classes d'implémentation (POJO Wrapper) et de proxy pour les technologies distribuées (EJB, Services Web , RMI , JMS,) + Contexte abstrait pour JNDI , ...
Spring ORM	Support abstrait pour les technologies de mapping objet/relationnel (ex: TopLink , Hibernate , iBatis, JDO, JPA ...)
Spring Web	WebApplicationContext , support pour le multipart/UploadFile, points d'intégration pour des frameworks STRUTS , JSF, ...
Spring Web MVC (optionnel mais recommandé)	Version "Spring" pour un framework Web/MVC. Ce framework est "simple/extensible" et "IOC". Vis à vis du concurrent "JSF" , c'est visuellement plus pauvre mais c'est moins exclusif , c'est plus flexible , modulaire et ça peut s'associer à d'autres technologies complémentaires (ex : thymeleaf). <u>NB</u> : Spring web mvc est très souvent utilisé comme alternative possible à JAX-RS pour développer des services web "REST"

==> plusieurs petits "*spring-moduleXY.jar*" complémentaires (souvent précisés via "maven").

Modules complémentaires pour Spring (extensions facultatives) :

Extensions fondamentales :

Extensions Spring	Contenus / spécificités
Spring- security	Extension très utile pour gérer la sécurité JEE (roles , authentification , ...)
Spring Data	Dao automatiques (pouvant être basés sur JPA ou bien MongoDB ou ...). Très bonne extension. Attention aux différences "spring4 , spring5"

Extensions secondaires :

Extensions Spring	Contenus / spécificités
Spring Web flow	Extension pour bien contrôler la navigation et rendre abstraite l'IHM (paramétrages xml des états , transitions, ...)
Spring Batch	prise en charge efficace des traitements "batch" (job , ...)
Spring Integration	Extensions pour SOA (fonctionnalités d'un mini ESB , EIP, ...)

6. Configurations Spring – vue d'ensemble

6.1. Historique et évolution

Versions de Spring	Possibilités au niveau de la configuration
Depuis Spring 1.x	Configuration entièrement XML (avec entête DTD) <bean>
Depuis Spring 2.0	Configuration XML (avec entête XSD) + .properties
Depuis Spring 2.5	Annotations spécifiques à Spring (@Component, @Autowired, ...)
Depuis Spring 3.0	Compatibilité avec annotations DI (@Inject, @Named)
Depuis Spring 4.0	Java Config (@Configuration, ...) et Spring boot 1.x (avec ou sans @EnableAutoConfiguration)
Depuis Spring 5.0	restructuration interne pour mieux intégrer java 8,9,10 et un début d'architecture asynchrone et réactive (Netty, WebFlux,) Spring Boot 2.x bien au point
Depuis Spring 6.0	Java 17 au minimum, Spring Boot 3.x, package jakarta.persistence.*

Spring (historique et évolution)

Complex et lourd

J2EE 1.x et EJB 1 & 2

JEE 5 et EJB 3.0

@Entity (JPA 1.0), @EJB

JEE 6 et EJB 3.1

JPA 2.0, @Named, @Inject

JEE 7, EJB 3.2, CDI

JAX-RS 2 (WS-REST)

Jakarta EE 9 et 10,
WebProfile, MicroProfile

Simple et efficace (le printemps) 2003-2007 environ

Spring 1.x

Spring 2.5

@Component, @Autowired,
@Transactional

Spring 3.x

Spring 4.x et 5.x

Spring-boot, @Configuration,
Spring-data, @RestController, ...

Spring 6.x, java >= 17

2006-2009 environ

2009-2013 environ

2013-2019 environ

Depuis 2023

Evolutions récentes importantes :

La base du langage java (le jdk et java-se) a été créé par Sun-Microsystem et est maintenant maintenu/géré par l'entreprise Oracle .

Evolution du standard (relatif) JEE :

<i>Époque, versions</i>	<i>Propriétaire</i>	<i>Namespace et packages</i>
1999-2009 : Java EE <=6	SUN	http://java.sun.com
2009-2013 : transition	Oracle	http://xmlns.jcp.org
2013-2018 : Java EE 7		<code>javax.persistence.*</code> pour JPA
2018-2020 : transition "EE8" après 2020 : Jakarta EE >=9	Jakarta (fondation eclipse, ...)	https://jakarta.ee , https://jakarta.ee/xml/ns/persistence <code>jakarta.persistence.*</code> pour JPA

Depuis 2018, le standard JEE n'appartient plus à Oracle .

Renommé "Jakarta EE" , le paquet d'api standard "EE" est maintenant géré par la communauté opensource "eclipse/jarkata" .

Depuis la version jakarta EE9 , les namespaces des fichiers de configurations (ex : persistence.xml) et les packages ont changés.

Parmi les évolutions importantes de JakartaEE , on peut noter :

- le support de `java>=11` depuis Jakarta EE 9.1
- la suppression d'anciennes choses obsolètes depuis jakartaEE10
- les api du paquet "Micro-profile" (pour API REST et Cloud)

Depuis l'origine de JEE , Le Framework Spring fait l'effort d'être au plus proche du standard JEE. **Ainsi depuis Spring 6 et SpringBoot 3 , les dépendances "JPA" sont à la sauce "jakarta"** (avec un package en `jakarta.persistence` et non plus `javax.persistence`).

Correspondance de versions :

Spring 6 et SpringBoot3 nécessitent :

`java >=17` (soit `java17` ou `java21`)

Maven ≥ 3.5 ou bien Gradle≥ 7.5

Servlet-Api ≥ 5 (ex : Tomcat 10.0 , Jetty 11, Undertow 2.2)

6.2. Avantages et inconvénients de chaque mode de configuration

Mode de config	Avantages	Inconvénients
XML	<ul style="list-style-type: none"> -Très explicite - Assez centralisé tout en étant flexible (import) . - utilisation possible de fichiers annexes ".properties" 	<ul style="list-style-type: none"> -Verbeux , plus à la mode - à maintenir / ajuster (si refactoring) - délicat (oblige à être très rigoureux "minuscules / majuscules" , noms des packages , namespaces XML , ...)
Annotations au sein des composants (@Autowired, ...)	<ul style="list-style-type: none"> - très rapide / efficace - suffisamment flexible (component-scan selon packages , @Qualifier , ...) - réajustement automatique en cas de refactoring (sauf component-scan) . 	<ul style="list-style-type: none"> - configuration dispersée dans le code de plein de composants - pour nos composants seulement (avec code source)
Classes de configuration "java" (@Configuration , ...)	<ul style="list-style-type: none"> -Très explicite - Assez centralisé tout en étant flexible (@Import) . - Auto complétion java et détection des incompatibilités (types , configurations non prévues, ...) - utilisation possible de fichiers annexes ".properties" pour les paramètres amenés à changer - à la mode ("hype") - configuration automatique / intelligente possible (selon classpath, env, ...) 	<ul style="list-style-type: none"> - nécessite une compilation de la configuration java (heureusement souvent automatisée par maven ou autre)

Spring (vue d'ensemble sur formats de configuration)

Anciens projets

(2004-2008 environ)

Configuration entièrement XML

(<bean ...><property ...> ...) et souvent accompagné de hibernate (.hbm.xml)

Projets commencés dans les années 2009-2014 environ

Configuration mixte XML + annotations
(@Autowired , @Component) et souvent accompagné de JPA (@Entity , @Id)

(proche standard JEE / @Inject , ...)

Projets récents

(depuis 2015 environ)

Configuration mixte javaConfig
(@Configuration, ...) + annotations
(@Autowired, ...) ou éventuellement auto-configuration avec application.properties et souvent avec Spring-boot , Spring-data , Spring-security et Spring-mvc (pour WS-REST)

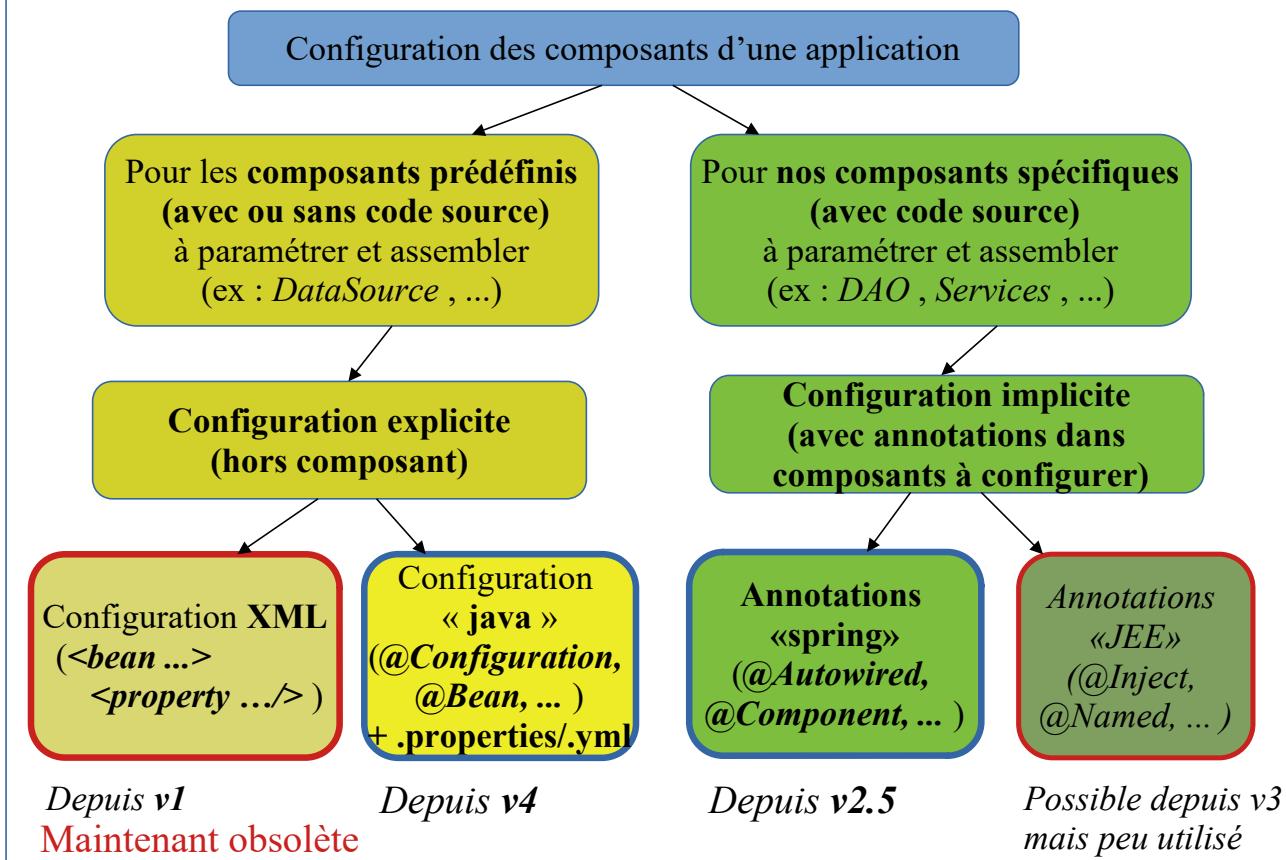
II - Configuration de Spring et tests essentiels

1. Vue d'ensemble sur config Spring

1.1. Complémentarité nécessaire / configuration mixte

- Les annotations `@Component`, `@Autowired`, sont très pratiques pour configurer des relations entre composants (injection de dépendances) mais elles **ne peuvent être utilisées qu'au niveau de nos propres composants** (car il faut avoir un contrôle total sur le code source).
- Une **configuration XML (ancienne)** ou bien une **configuration "java config" (moderne)** permet de configurer des composants génériques (ex : `DataSource`, `TransactionManager`,) dont on ne dispose pas du code source .
- Dans tous les cas, on s'appuie sur des **fichiers annexes** au format `".properties"` ou bien `".yml"` pour simplifier l'édition de quelques paramètres clefs susceptibles de changer (ex : url JDBC, username, password, ...)

Spring (configurations possibles)



1.2. Démarrages possibles depuis spring 4,5,6

Depuis méthode main() dans une application « standalone »	ApplicationContext <i>springContext</i> = new AnnotationConfigApplicationContext (<i>My AppConfig.class</i> , <i>ConfigSupplementaire.class</i>) ; Cxy c = (Cxy) <i>springContext.getBean("idBeanXy")</i> ; //ou bien c = <i>springContext.getBean(Cxy.class)</i> ;
Depuis test unitaire (Junit + spring-test)	@ExtendWith (SpringExtension.class) @ContextConfiguration(classes={My AppConfig.class}) public class TestCxy { @Autowired private Cxy c ; //+ méthodes prefixées par @Test }

+ tous les anciens démarrages possibles en vieilles syntaxes XML (*voir doc complémentaires sur anciennes versions*)

+ démarrage possible en mode "web" , dans conteneur Web tel que tomcat (*voir chapitre web*)

+ tous les nouveaux démarrages possibles via **spring-boot** (*voir chapitre spring-boot*).

1.3. Vue d'ensemble sur les aspects de la configuration spring

profiles (@Profile , ...)	variantes de configuration activées ou pas au démarrage de l'application
@Qualifier	Qualificatifs facultatifs que l'on peut donner à plusieurs composants spring de même type (de même classe ou même interface) pouvant coexister en même temps . Ceci permet d'injecter si besoin une version bien précise

1.4. Configuration structurée (properties , import , profiles)

Spring (paramétrages indirects dans fichiers ".properties")

Quelque soit la version de Spring, en partant d'une configuration globale explicite ordinaire (xml/bean ou bien java/@Configuration), il est possible de récupérer certaines valeurs variables (de paramètres clefs) dans un fichier annexe au format "**.properties**"

Ceci s'effectue techniquement via
"*PropertySourcesPlaceholderConfigurer*" ou un équivalent .

Configuration principale
(`context.xml` ou bien
`MyAppConfig.class`)
avec
 `${database.url}` ,
 `${database.username}`,
....

*configuration technique et stable
(quelquefois complexe)*

myApp.properties

```
database.url=jdbc:mysql://...
database.username=user1
database.password=pwd1
....
```

*sous configuration (indirecte)
potentiellement variable et facile
à modifier (claire, simple).*

Spring (Configuration structurée via "import")

`applicationContext.xml`
ou bien
`applicationConfig.class`

import

`securityConfig.xml/.class`

import

`dataSourceConfig.xml/.class`

import

`jpaConfig.xml/.class`

ou

`domainAndPersistenceConfig.xml/.class`

import

`webMvcConfig.xml/.class`

Profiles (Variantes de configurations) depuis Spring4

@Profile({"!test"})

ou bien

@Profile({"jta", "test"})

au dessus de variantes

de **@Bean** dans

@Configuration

`context.getEnvironment().setActiveProfiles(...);`

ou bien

`springBootApp.setAdditionalProfiles(...);`

ou bien

@ActiveProfiles(profiles = {"test", "jta"})

au dessus d'une classe de test (@RunWith, ...)

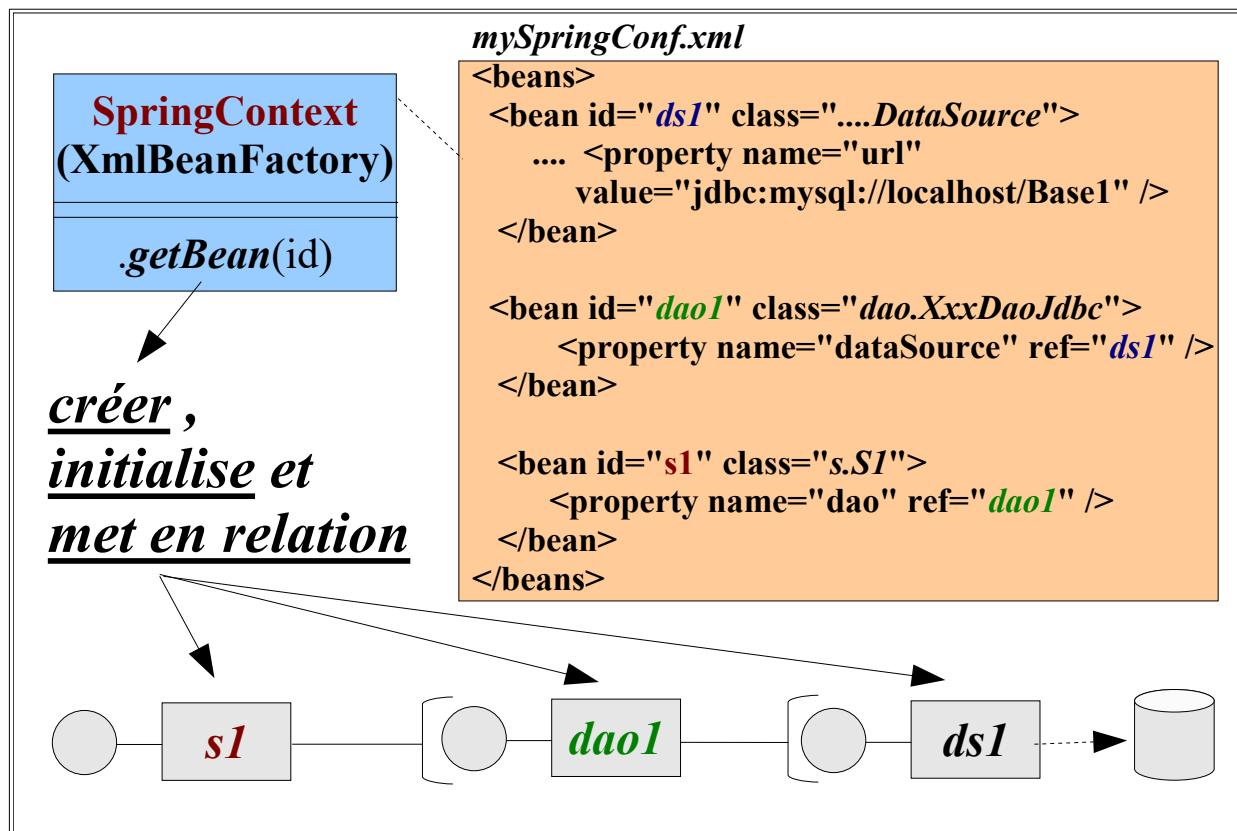
2. Ancienne configuration Xml de Spring (aperçu)

Configurer Spring avec des fichiers xml est aujourd'hui un peu obsolète .

La configuration Xml de Spring n'est aujourd'hui qu'à étudier et utiliser que pour maintenir des anciens projets (des années 2005-2015) .

---> le chapitre "configuration Spring XML" a maintenant été déplacé dans un document annexe.

Voici tout de même un rapide micro-aperçu :



NB :

- Bien que obsolète , il est encore possible d'utiliser une configuration xml avec une version récente de spring (ex : spring 6) .
- Il est possible de mixer dans tous les sens différents types de configurations (ex : configuration xml englobant une sous configuration java , ...)
- Un petit exemple complet d'application spring configurée en xml est accessible au sein du référentiel git https://github.com/didier-tp/spring6_2024.git (projet "oldXmlSpringApp" de la partie "tp")

3. Configuration IOC Spring via des annotations

Depuis la version 2.5 de Spring, il est possible d'utiliser une configuration IOC paramétrée par des annotations directement insérées dans le code java à la place d'une configuration entièrement XML.

Pour cela, Spring utilise essentiellement les annotations suivantes :

`@Component`, `@Service`, `@Repository`, `@RestController`, `@Autowired`, `@Qualifier`, ...

NB :

- Ces annotations doivent être placées au bon endroit dans une classe java applicative et nécessitent donc un accès au code source des composants à paramétrer
- Ce mode de configuration de l'injection de dépendance(`@Component` + `@Autowired`) est le plus simple/rapide à mettre en oeuvre
- La configuration complète d'une application est très souvent un mixte "java-config (`@Configuration`/`@Bean`) + annotations (`@Component`/`@Autowired`)"

3.1. Annotations (stéréotypées) pour composant applicatif

exemple : `XyDaoImpl.java`

```
package tp.dao;

import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;
import tp.entity.Xy;

@Component("myXyDao")
public class XyDaoImpl implements XyDao {

    public Xy getXyByNum(long num) {
        Xy xy = new Xy();
        xy.setNum(num);
        xy.setLabel("?? simu ??");
        return xy;
    }
}
```

dans cet exemple, l'annotation `@Component()` marque (ou stéréotype) la classe Java comme étant celle d'un *composant pris en charge par Spring*. D'autre part, la valeur facultative "myXyDao" correspond à l'ID qui lui est affecté. (*l'id par défaut est le nom de la classe avec une minuscule sur la première lettre* : "xyDaoImpl" au sein de cet exemple).

NB: Les stéréotypes **@Repository**, **@Service** et **@Controller** (qui héritent tous les 3 de **@Component**) sont avant tout destinés à marquer le type des composants dans une architecture n-tier. Ceci permet alors d'automatiser certains traitements en tenant compte de ces stéréotypes que l'on peut découvrir/filtrer par introspection .

On peut éventuellement utiliser ces annotations pour renseigner l'id précis d'un composant Spring.

@Component	Composant spring quelconque
@Repository	Composant d'accès aux données (DAO)
@Service	Service métier (alias business service) avec transactions
@Controller	Composant de contrôle IHM (coordinateur, ...)
@RestController	Composant de contrôleur de Web Service REST

3.2. Autres annotations ioc (@Required , @Autowired , @Qualifier)

@Required (rare) (à placer au dessus d'une méthode d'injection ou d'une propriété privée)	Pour vérifier dès le début (initialisation du contexte Spring et ses composants) qu'une injection a bien été effectuée . Si la valeur de la référence est restée à null --> exception dès l'initialisation plutôt qu'en cours d'exécution du programme.
@Autowired	Pour demander une auto-liaison par type (injections de dépendances automatiques et implicites en fonction des correspondances de type). Par défaut <code>@Autowired(required=true)</code>
@Qualifier	Permet de marquer une injection Spring avec un qualificatif / nom de variante (ex: "test" ou "prod" ou ...) dans le but de paramétriser plus finement les auto-liaisons (éventuel filtrage selon le qualificatif attendu)

3.3. @Autowired (fondamental)

Exemple (assez conseillé) avec **@Autowired**

```
@Service() //id par défaut = serviceXyImpl
public class ServiceXyImpl implements ServiceXy {

    @Autowired
    // injectera automatiquement l'unique composant Spring configuré
    // dont le type est compatible avec l'interface précisée.
    private XyDao xyDao;

    public Xy getXyByNum(long num) {
        return xyDao.getXyByNum(num);
    }
}
```

- Sans **@Autowired** , `this.xyDao` resterait tout le temps à null (valeur par défaut en java).
- Grace à **@Autowired** , Spring initialise automatiquement la valeur de `this.xyDao` en y plaçant une référence sur un composant spring existant compatible avec l'interface `XyDao` .

NB : il est éventuellement possible de placer @Autowired sur un setter :

```
@Service() //id par défaut = serviceXyImpl
public class ServiceXyImpl implements ServiceXy {
    private XyDao xyDao = null;

    @Autowired
    public void setXyDao(XyDao xyDao) {
        this.xyDao = xyDao;
    }
    ...
}
```

Injection facultatives (non fondamentales) :

Par défaut, @Autowired demande à effectuer une injection de dépendance absolument nécessaire et l'application Spring ne démarre alors pas bien (avec message d'erreur explicite) en cas de problème (pas de composant possible à injecter ou bien ambiguïté à résoudre).

Via l'éventuel paramètre **required=false**, on peut demander une injection facultative (qui sera réalisée ou pas en fonction du contexte / des possibilités) :

```
...
@Autowired(required=false)
private Ixxx choseFaculativeAinjecter; //null par défaut.

methodeTraitementQuiVaBien(){
    if(this.choseFaculativeAinjecter!= null){
        //code qui va bien
    }
}
```

Variantes pour l'injection de dépendances :

```
import javax.annotation.Resource; // standard java/jee
import javax.inject.Inject; // standard java/jee
import org.springframework.beans.factory.annotation.Autowired; // spécifique spring
```

- **@Autowired** effectue une injection par correspondance de type, et pour une correspondance de nom on a alors besoin du complément **@Qualifier**
- **@Resource** effectue une injection par correspondance de nom (si précisé), sinon par correspondance de type (si le nom n'est pas précisé)
- **@Inject** nécessite un ajout dans pom.xml (javax.inject) et est alors interprété par spring comme un équivalent de **@Autowired**

```
@Resource("nomComposant_a_injecter")
//@Inject @Qualifier("nomComposant_a_injecter")
private Ixx choseAInjecterParCorrespondanceDeNom;

//@Inject
@Autowired
private Iyy choseAInjecterParCorrespondanceDeType;
```

3.4. Paramétrage des @ComponentScan de "Spring"

En organisant bien les packages java de la façon suivante :

xxx.ifc.dao.DaoXy (interface)

xxx.impl.dao.v1.DaoXyImpl1 (classe d'implémentation du Dao en version 1 avec @Component)

xxx.impl.dao.v2.DaoXyImpl2 (classe d'implémentation du Dao en version 2 avec @Component)

on peut ensuite paramétriser alternativement une configuration java Spring de l'une des 2 façons suivantes :

@Configuration

```
@ComponentScan(basePackages={"xxx.impl.dao.v1","org.mycontrib.generic"})
public class XyzConfig {
```

...

}

ou bien

@Configuration

```
@ComponentScan(basePackages={"xxx.impl.dao.v2","org.mycontrib.generic"})
public class XyzConfig {
```

...

}

ceci fait que une seule des deux versions (v1 ou v2) est prise en charge par Spring et donc candidate à une injection paramétrée via @Autowired .

Il n'y a alors plus d'ambiguïté au niveau de

```
@Autowired //ou @Inject
private DaoXy xyDao ;
```

NB : @ComponentScan comporte plein de variantes syntaxiques (include , exclude , ...)

Autre solution élégante pour choisir entre l'alternative v1 et v2

---> utiliser des profiles spring au niveau des composants :

```
@Component
```

```
@Profile({"v1"})
```

```
class DaoXyYImpl1 implements DaoXY {
```

....

```
}
```

```
@Component
@Profile({"v2"})
class DaoXyImpl2 implements DaoXY {
...
}
```

et

avec Spring_framework :

```
System.setProperty("spring.profiles.active", "v1,profileComplementaireA");
ApplicationContext contextSpring = new AnnotationConfigApplicationContext(XyConfig.class);
ou bien (avec SpringBoot) :
SpringApplication app = new SpringApplication(MySpringBootApplication.class);
app.setAdditionalProfiles("v1","profileComplementaireA") ;
ConfigurableApplicationContext context = app.run(args);
//ou autre façon de démarrer (ex : @SpringBootTest et @ActiveProfiles({"v1","pA"}))
```

--> selon le profile "v1" ou bien "v2" sélectionné au démarrage d'un test ou de l'application spring , une seule des 2 versions *DaoXyImpl1* ou *DaoXyImpl2* sera prise en charge par Spring et donc candidate à une injection paramétrée via @Autowired .

3.5. Configuration minimum de démarrage (appli. Spring)

```
...
@Configuration
@ComponentScan(basePackages = {"tp.appliSpring"})
//avec package tp.appliSpring comportant sous packages tp.appliSpring.dao, .entity, .service, ...
public class SimpleConfig {
```

```
...
public class BasicSpringApp {
    public static void main(String[] args) {
        ApplicationContext springContext =
            new AnnotationConfigApplicationContext(SimpleConfig.class);
        ServiceXy serviceXy = springContext.getBean(ServiceXy.class);
        System.out.println(serviceXy.methodeQuiVaBien());
    }
}
```

3.6. @Qualifier (pour variantes qui peuvent coexister)

```
import org.springframework.beans.factory.annotation.Qualifier;
```

```
@Component @Qualifier("byCreditCard")
class PaymentByCreditCard implements Payment {
...
}
```

```
@Component @Qualifier("byCash")
class PaymentByCash implements Payment {
...
}
```

```
@Component
class ServiceXyDelegatingPayment implements ... {
    @Autowired @Qualifier("byCreditCard")
    private Payment paiementParCarteDeCredit ;

    @Autowired @Qualifier("byCash")
    private Payment paiementEnLiquide ;

    public void payer(double montant){...}
}
```

@Qualifier est surtout pratique pour **injecter différentes variantes pouvant coexister** en même temps et non pas des **alternatives exclusives** .

Attention (source de confusion possible):

- La variante spring de **@Qualifier("..."")** est une annotation simple à utiliser et permettant de préciser la version de l'on souhaite injecter via une correspondance de nom ou de qualificatif.
- *Au sein de la technologie java "DI/CDI" concurrente vis à vis de spring , il existe une autre variante du @Qualifier (javax.inject.Qualifier) qui est bien différente et bien plus complexe à utiliser (javax.inject.Qualifier est une métà annotation qui sert à construire de nouvelles annotations spécifiques telles que @ByCash ou bien @ByCreditCard) .*

Variante comportementale importante:

Si un composant spring est déclaré sans `@Qualifier` et donc seulement avec `@Component` ou équivalent , c'est alors comme si il avait un qualificatif par défaut correspondant à son nom logique (valeur entre les parenthèses de `@Component` ou bien `nomClasseJavaAvecMinusculeSurPremiereLettre`) .

```
@Component //defaultName = paymentByCreditCard
class PaymentByCreditCard implements Payment {
...
}
```

```
@Component("paiementEnLiquide")
class PaymentByCash implements Payment {
...
}
```

```
@Component
class ServiceXyDelegatingPayment implements ... {
    @Autowired @Qualifier("paymentByCreditCard")
    private Payment paiementParCarteDeCredit;

    @Autowired @Qualifier("paiementEnLiquide")
    private Payment paiementEnLiquide;

    public void payer(double montant){...}
}
```

3.7. rares `@Scope` et `@Lazy`

Par défaut chaque composant spring (déclaré via `@Component` ou autre) est **construit/initialisé dès le démarrage de l'application** et est instancié une seule fois (selon le design pattern "singleton").

Dans des cas rares, on peut placer `@Lazy` à coté de `@Component` et dans ce cas l'instance de la classe ne sera construite par Spring que plus tard au moment de l'appel à `.getBean(...)` .

Si l'on appelle plusieurs fois `.getBean("...")` avec le même nom logique de composant, ou bien si l'on injecte plusieurs fois ce composant à différents endroits, on récupère alors par défaut toujours la même valeur (référence vers une unique instance / singleton) .

Cela vient du fait que `@Scope("...")` comporte par défaut la valeur "singleton" .

Valeurs possibles pour `@Scope("...")`:

ConfigurableBeanFactory.SCOPE_SINGLETON="singleton"

ConfigurableBeanFactory.SCOPE_PROTOTYPE="prototype" (un `new` suite à chaque `.getBean(...)`)

WebApplicationContext.SCOPE_REQUEST ="request" (pour composant web seulement)

WebApplicationContext.SCOPE_SESSION="session" (pour composant web seulement)

4. Cycle de vie , @PostConstruct , @PreDestroy

Cycle de vie d'un composant pris en charge par Spring :

- 1) instanciation (appel au constructeur)
- 2) injections de dépendances (selon @Autowired)
- 3) appel à la méthode préfixée par **@PostConstruct** (si elle existe)
- 4) utilisation normale du composant spring
- 5) appel à la méthode préfixée par **@PreDestroy** (si elle existe) lors d'un arrêt (pas brutal) du contexte spring
- 6) éventuel appel à la méthode finalize() (si elle existe) sur l'instance java

```
//import javax.annotation.PostConstruct; import javax.annotation.PreDestroy;
import jakarta.annotation.PostConstruct; //Depuis Spring6 et springBoot3
import jakarta.annotation.PreDestroy;

public class XxxService
{
    @Autowired
    private IZzz zzObj ;

    @Autowired
    private IYyy yyObj ;

    private String v ; //valeur_a_initialiser_au_plus_tôt !

    public XxxService(){
        //NB: Le constructeur est déclenché avant la gestion
        // des @Autowired
        //donc zzObj et yyObj sont à null
        //et ne sont pas encore utilisables
        //dans le ou les constructeur(s)
    }

    @PostConstruct
    public void initBean() {
        //premier endroit où this.zzObj et this.yyObj ne sont normalement plus à null
        this.v = this.zzObj.recupValeur() ; ...
    }

    @PreDestroy
    public void cleanUp() {
        System.out.println("cleanUp before end of Spring");
    }
}
```

5. Injection par constructeur (assez conseillé)

@Component

```
public class Cx {  
    public String ma() { return "abc"; }  
}
```

@Component

```
public class Cz {  
    public String mb() { return "def"; }  
}
```

@Component

```
public class Cy {  
    private Cx x;  
    private Cz z;  
  
    // @Autowired //explicit or implicit if just one constructor  
    public Cy(Cx x, Cz z) {  
        this.x=x; this.z=z;  
    }  
  
    public String mab() {  
        return x.ma() + "-" + z.mb();  
    }  
}
```

@Configuration

```
@ComponentScan(basePackages = {"org.mycontrib.backend.demo"})  
public class LittleConfig {  
}
```

```
public class LittleDemoApp {  
    public static void main(String[] args) {  
        ApplicationContext context =  
            new AnnotationConfigApplicationContext(LittleConfig.class);  
        Cy y = context.getBean(Cy.class);  
        System.out.println(y.mab());  
    }  
}
```

6. "Java Config" mieux qu'ancienne config xml

Depuis "Spring 4" , l'extension "*java config*" est maintenant intégrée dans le cœur du framework et il est maintenant possible de **configurer une application spring par des classes java** spéciales (dites de configuration").

NB : une configuration mixte "xml + java-config" est éventuellement possible.

NB : Depuis "Spring 5" et "Spring-Boot" , la configuration "java-config" est devenue la configuration de référence dans l'écosystème "spring moderne" et a complètement éclipsé l'ancienne configuration xml.

Premiers avantages d'une configuration explicite java (par rapport à une configuration xml) :

- Auto complétion java et détection des incompatibilités (types , configurations non prévues, ...)
- Héritage possible entre classes de configuration (générique, spécifique, ...)
- configuration intelligente (selon classpath, selon env, ...)

7. Java Config (Spring) en fonction du contexte

Au sein d'une application uniquement basée sur "Spring-framework"	La configuration " java config " sera alors la configuration principale (<i>point de démarrage</i> de la fonction main() ou équivalent web)
Au sein d'une application basée sur SpringBoot	La configuration " java config " ne sera utilisée qu'en tant que configuration additionnelle pour des cas particuliers ou bien annexes (ex : sécurité , composants utilitaires, ...)

- **La configuration "java config" (explicite et plus complexe qu'une simple utilisation de @Component et @Autowired) ne sera généralement utilisée que lorsque l'on ne peut pas (ou que l'on ne souhaite pas) modifier le code code source des composants à paramétriser.**
- **La configuration "java" explicite (qu'il faut recompiler en cas de changement) aura souvent intérêt à analyser des fichiers ".properties" plus faciles à modifier et paramétrier .**

NB :

- Les exemples de configuration de ce chapitre ne sont à considérer que comme des exemples de configurations possibles (à adapter en fonction du contexte) !!!
- Certains points avancés seront exposés dans une annexe

8. Config java élémentaire

8.1. Exemple1: DataSourceConfig :

```
package tp.myapp.config;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Configuration
public class DataSourceConfig {

    @Bean(name="myDataSource") //by default beanName is same of method name
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/minibank_db_ex1");
        dataSource.setUsername("root");
        dataSource.setPassword("root");//"root" ou "formation" ou "..."
        return dataSource;
    }
}
```

NB :

- cette classe de configuration "**DataSourceConfig**" sert à configurer un composant spring applicatif basé sur la classe "**DriverManagerDataSource**" prédéfinie dans **spring-jdbc** (*jar téléchargé par maven*) et implémentant l'interface **DataSource** standard du langage java.
- Une version plus élaborée (analysant un fichier .properties) sera présentée ultérieurement.

Comportement fondamental :

- **@Configuration** sert à marquer une classe **XyzConfig** comme une **classe de configuration** qui sert à fabriquer des composants (**@Bean**) issus d'autres classes java (*dont on ne dispose pas toujours du code source*).
- **Chaque méthode préfixée par @Bean sert à fabriquer et initialiser une instance** d'une classe java **qui sera vu comme un composant spring** (injectable par **@Autowired** et accessible via **springContext.getBean(...)**)
- **Par défaut l'id (nom logique) du composant généré sera le nom de la méthode qui a servi à le fabriquer** (ex : "dataSource" dans l'exemple ci-dessus) **et @Bean(name="myDataSource") sert à spécifier un nom (id) spécifique .**
- Un composant spring peut éventuellement avoir plusieurs noms (un nom principal et des alias) : **@Bean(name = { beanName, alias1 , alias2 })**.

8.2. Utilisations possibles (ici sans spring-boot):

Dans main() :

```
ApplicationContext context =  
    new AnnotationConfigApplicationContext(DataSourceConfig.class,  
                                         DomainAndPersistenceConfig.class);  
DataSource ds = context.getBean(DataSource.class);  
...
```

Possible mais très rare : dans springContext.xml (pour config java intégrée dans config xml):

```
<context:annotation-config /> <!-- pour interprétation de @Configuration , @Bean -->  
<bean class="tp.myapp.config.DataSourceConfig"/>
```

Dans spring test (ici sans spring-boot):

```
// @RunWith(SpringJUnit4ClassRunner.class) // si JUnit4  
@ExtendWith(SpringExtension.class) si JUnit5/jupiter  
  
//@ContextConfiguration(locations="/springContextOfModule.xml") // si xml config  
@ContextConfiguration(classes={tp.myapp.config.DataSourceConfig.class, ...}) //java config  
// ou bien @SpringBootTest(classes= {MySpringBootApplication.class}) si spring-boot  
  
public class TestXy {  
    @Autowired  
    private .... ;  
  
    @Test  
    public void testXy(){ .....  
    }  
}
```

8.3. Avec placeHolder et fichier ".properties"

src/main/resources/**datasource.properties** (exemple) :

```
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:~/mydb
spring.datasource.username=sa
spring.datasource.password=
```

NB : bien que "sans spring-boot" cet exemple reprend volontairement les mêmes noms classiques de propriétés que **application.properties**

DataSourceConfig.java

```
...
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;
@Configuration
//équivalent de <context:property-placeholder location="classpath:datasource.properties" /> :
@PropertySource("classpath:datasource.properties")
public class DataSourceConfig {

    @Value("${spring.datasource.driverClassName}")
    private String jdbcDriver;

    @Value("${spring.datasource.url}")
    private String dbUrl;

    @Value("${spring.datasource.username}")
    private String dbUsername;

    @Value("${spring.datasource.password}")
    private String dbPassword;

    /*
    //Configuration additionnelle anciennement nécessaire au sein des anciennes versions de spring
    @Bean
    public static PropertySourcesPlaceholderConfigurer
        propertySourcesPlaceholderConfigurer(){
        return new PropertySourcesPlaceholderConfigurer();
        //pour pouvoir interpréter ${} in @Value()
    }
    */

    @Bean(name="myDataSource")
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(jdbcDriver);
        dataSource.setUrl(dbUrl);
        dataSource.setUsername(dbUsername);
        dataSource.setPassword(dbPassword);
        return dataSource;
    }
}
```

NB: Dans le cas (*très fréquent d'une configuration automatique avec @EnableAutoConfiguration ou bien @SpringBootApplication*) , pas besoin d'expliciter
`@PropertySource("classpath:application.properties")` car c'est également déjà configuré automatiquement .

==> **et donc dans la plupart des cas juste besoin de :**

- placer les propriétés au bon endroit (dans le fichier **application.properties** ou **application.yml** ou ...)
- référencer à accès à ces propriétés via `@Value("${xx.yy.property-name}")`

8.4. @Value avec Spring-EL

NB :

- Au sein de `@Value()` , les syntaxe en `${...}` servent à évaluer des expressions dont la formulation est spécifique à **Spring-EL** (spring Expression Language) .
- Des valeurs par défaut peuvent être placées après un ":"

`@Value("${xx.yy.property-name}")` ou bien

`@Value("${xx.yy.property-name : default_value_if_no_present_in_application_properties}")`

Syntaxes avec valeurs par défaut :

```
@Value("${some.key:my default value}")
private String stringWithDefaultValue;
```

```
@Value("${some.key:true}")
private boolean booleanWithValue;
```

```
@Value("${some.key:42}")
private int intValue;
```

```
@Value("${some.key:one,two,three}")
private String[] stringArrayWithDefaults;
```

8.5. Quelques paramétrages (avancés) possibles :

```
@Bean(initMethodName="init") , @Bean(destroyMethodName="cleanup")
//sachant qu'on peut également placer @PostConstruct au dessus de init() et @PreDestroy .
```

```
@Bean(scope=DefaultScopes.PROTOTYPE) , @Bean(scope = DefaultScopes.SESSION)
//sachant que le scope par défaut est DefaultScopes.SINGLETON
```

8.6. injections de dépendances entre @Bean

- La configuration complète d'une application est souvent répartie dans plusieurs classes de `@Configuration` .
- Un `@Bean` configuré dans une première classe de `@Configuration` peut éventuellement être injecté et utilisé par un autre `@Bean` d'une autre classe de `@Configuration` .

Exemple :

`@Configuration`

```
public class XyzConfig {

    @Bean /* @Qualifier("cx1") */
    public Ix cx1() {
        Ix cxBean = new Cx1();
        return cxBean;
    }

    @Bean /* @Qualifier("cx2") */
    public Ix cx2() {
        Ix cxBean = new Cx2();
        return cxBean;
    }

    @Bean
    public Iy cy() {
        Iy cyBean = new Cy();
        return cyBean;
    }

    @Bean
    public Iz cz(@Qualifier("cx1") Ix x, Iy y) {
        /* Iz czBean = new Cz(x,y); // si possible */
        Iz czBean = new Cz();
        czBean.setX(x);
        czBean.setY(y);
        return czBean;
    }
}
```

NB :

- les éléments à injecter (nécessaire à la construction d'un composant dépendant d'autres composants) doivent être placés en tant que paramètres d'entrées des méthodes préfixées par `@Bean`
- en cas d'ambiguité (si plusieurs versions possibles) , on pourra utiliser `@Qualifier`

8.7. Exemple concret: **JpaExplicitConfig**:

```

package tp.myapp.config;
import jakarta.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor;
import org.springframework.orm.jpa.vendor.Database;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

```

@Configuration

@EnableTransactionManagement() //"*transactionManager*" (not "txManager") is expected !!!

@ComponentScan(basePackages={"tp.myapp","org.mycontrib.generic"})
// for interpretation of @Component , @Controller , ... for @Autowired, @Inject ,...

//@EntityScan(basePackages={"tp.myapp.entity"}) //to find and interpret @Entity, ...

```

public class JpaExplicitConfig {

    // JpaVendorAdapter (Hibernate ou OpenJPA ou ...)
    @Bean
    public JpaVendorAdapter jpaVendorAdapter() {
        HibernateJpaVendorAdapter hibernateJpaVendorAdapter
            = new HibernateJpaVendorAdapter();
        hibernateJpaVendorAdapter.setShowSql(false);
        hibernateJpaVendorAdapter.setGenerateDdl(false);
        hibernateJpaVendorAdapter.setDatabase(Database.MYSQL);
        //hibernateJpaVendorAdapter.setDatabase(Database.H2);
        return hibernateJpaVendorAdapter;
    }

    // EntityManagerFactory
    @Bean(name= { "entityManagerFactory", "myEmf" , "otherAliasEmf" } )
    public EntityManagerFactory entityManagerFactory(
        JpaVendorAdapter jpaVendorAdapter, DataSource dataSource) {
        LocalContainerEntityManagerFactoryBean factory
            = new LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(jpaVendorAdapter);
        factory.setPackagesToScan("tp.myapp.entity");
        factory.setDataSource(dataSource);

        Properties jpaProperties = new Properties() ; //java.util
        jpaProperties.setProperty("javax.persistence.schema-generation.database.action",
            "drop-and-create") ; //à partir de JPA 2.1

        factory.setJpaProperties(jpaProperties) ;
        factory.afterPropertiesSet();
        return factory.getObject();
    }
}

```

```
// Transaction Manager for JPA or ...
@Bean(name="transactionManager") //("transactionManager" but not "txManager")
public PlatformTransactionManager transactionManager(
    EntityManagerFactory entityManagerFactory) {
    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory);
    return txManager;
}
}
```

NB : la configuration explicite ci-dessus est inutile en mode configuration automatique .

8.8. @Import explicites et implicites/automatiques

```
@Configuration
@Import(DomainAndPersistenceConfig.class)
//@ImportResource("classpath:/xy.xml")
@ComponentScan(basePackages={"tp.app.zz.web"})
@EnableWebMvc //un peu comme <mvc:annotation-driven />
public class WebMvcConfig {

    @Bean
    public ViewResolver mcvViewResolver(){
        InternalResourceViewResolver viewResolver =new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/view/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

NB (en mode configuration explicite):

@Import({SousPartieConfig1.class , SousPartie2Config.class}) est essentiellement utile **en mode "configuration explicite , non automatique"** pour importer/imbriquer des classes de configurations annexes/complémentaires rangées dans des packages génériques/utilitaires qui ne sont en règle générale pas directement liés au package principal de l'application courante.

NB (en mode configuration implicite/automatique):

Dans le cas (très fréquent aujourd'hui) d'une configuration automatique (avec **@SpringBootApplication** équivalent à peu près à **@EnableAutoConfiguration + @ComponentScan/main_package**), on considérera que le package principal de l'application est celui qui comporte la classe de démarrage de l'application (avec **@SpringBootApplication** et **main()**) .

Et dans ce cas toutes les classes de type **@Configuration** placées dans un des sous-packages du package principal de l'application seront alors automatiquement trouvées et activées (sans besoin de **@Import**) sauf si elles sont associées à des profils non activés au démarrage.

8.9. Profiles "spring" (variante de configuration)

Un **profil spring** est une **variante de configuration** (avec *nom libre* et *signification à définir*).

NB : certains profils peuvent être **exclusifs** (ex : "dev" ou bien "prod" , "withSecurity" ou bien "withoutSecurity") et d'autres peuvent être **complémentaires** (ex : "dev" et "withSecurity").

Déclarations/définitions des variantes :

@Profile({"!dev"}) //si profile "dev" pas activé

ou bien

@Profile({"dev"}) //si profile "dev" activé

à placer à coté de **@Configuration** (sur l'ensemble d'une classe de config)

ou bien au dessus d'une **variante de @Bean** dans **@Configuration**

ou bien à coté de **@Component** ou d'une annotation équivalente

@Configuration @Profile("dev") class XyConfigDev{ ... }	@Configuration class XyConfigDev{ ... @Profile("dev") @Bean public Xxx xxx(){ ... } }	@Component @Profile("dev") class InitDataSetInDev{ ... }
--	--	---

Sélection du ou des profile(s) à activer au démarrage de l'application ou des tests

scriptLancementAppliSpring.bat ou .sh

```
java ... -Dspring.profiles.active=dev,profileComplementaire2 ...
```

ou bien

```
System.setProperty("spring.profiles.active", "dev,profileComplementaire2");
```

```
ApplicationContext contextSpring = new AnnotationConfigApplicationContext(ConfigXy.class);
```

```
//dans méthode main() ou ailleurs avec Spring-framework
```

ou bien

```
springBootApp.setAdditionalProfiles("dev", "profileComplementaire2");
```

```
//au sein de la méthode main() avec SpringBoot
```

ou bien

```
@ActiveProfiles(profiles = {"dev", "profileComplementaire2"})
```

```
//au dessus d'une classe de test (avec @RunWith ou @ExtendWith)
```

NB : les points avancés de la configuration "java config" sont exposés dans une **annexe**

9. Tests "Junit4/5 + Spring" (spring-test)

Depuis la version 2.5 de Spring , il existe des annotations permettant d'initialiser simplement et efficacement une classe de Test JUnit avec un contexte (configuration) Spring.

Attention: pour éviter tout problème d'incompatibilité entre versions, il est souhaitable d'utiliser une version très récente de JUnit 4 ou 5 et spring-test .

Exemple de classe de Test de Service (avec annotations de JUnit4)

```
...
import static org.junit.Assert.assertTrue ;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

// nécessite spring-test.jar et junit 4 dans le classpath
@RunWith(SpringJUnit4ClassRunner.class)
//@ContextConfiguration(locations={"/mySpringConf.xml"}) //si config xml
//@ContextConfiguration(classes={XxConfig.class, YyConfig.class}) //java config
public class TestXy {

    @Autowired
    private IServiceXy service = null;

    @Test
    public void testXy(){
        assertTrue( ... );
    }
}
```

Adaptations pour JUnit 5 (jupiter)

```
...
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
...

// nécessite spring-test.jar et junit jupiter dans le classpath
@ExtendWith(SpringExtension.class)
```

```
@ContextConfiguration(classes={XxConfig.class, YyConfig.class}) //java config
public class TestXy {

@Autowired
private IServiceXy service = null;

@Test
public void testXy(){
    assertTrue( ... );
}

}
```

Cas particulier "Spring-Boot" :

Dans un **contexte "Spring + Spring-boot"** , il faut idéalement remplacer

```
@ContextConfiguration(classes={XxConfig.class, YyConfig.class})
par
@SpringBootTest(classes= {MySpringBootApplication.class})
```

Cas particulier pour certains tests de "DAO":

Un **Dao** est normalement utilisé par un service métier dont les méthodes sont transactionnelles.
Pour qu'une classe de **Test de dao** soit au plus près de la réalité , on peut éventuellement placer en elle des parties transactionnelles via **TransactionTemplate**.

Exemple :

```
@Autowired
TransactionTemplate txTemplate;
...
@Test
public void testCompteAvecOperationsEtAvecTransactionDeNiveauTest() {
    //phase1 (avec transaction1 comitée): insérer jeyx de données
    Long numCptA = txTemplate.execute(transactionStatus->{
        Compte cptA = daoCompte.save(new Compte(null,"compteAha",101.0));
        Operation op1 = new Operation(null,"achat 1" , -5.0 , new Date());
        op1.setCompte(cptA); daoOperation.save(op1);
        Operation op2 = new Operation(null,"achat 2" , -6.0 , new Date());
        op2.setCompte(cptA);daoOperation.save(op2);
        return cptA.getNumero();
    });
    System.out.println("numCptA="+numCptA);
    //phase2 (avec transaction2 comitée): relire les données stockées en base
    txTemplate.execute(transactionStatus->{
        Compte cptArelu = daoCompte.findById(numCptA).get();
        System.out.println("cptArelu="+cptArelu);
        //NB:ici en mode transactionnel , pas de lazy exception bien que LAZY et simple appel à findById
        for(Operation op: cptArelu.getOperations()) {
            System.out.println("\t op="+op);
        }
        assertTrue(cptArelu.getOperations().size()==2);
        return null;
    });
}
```

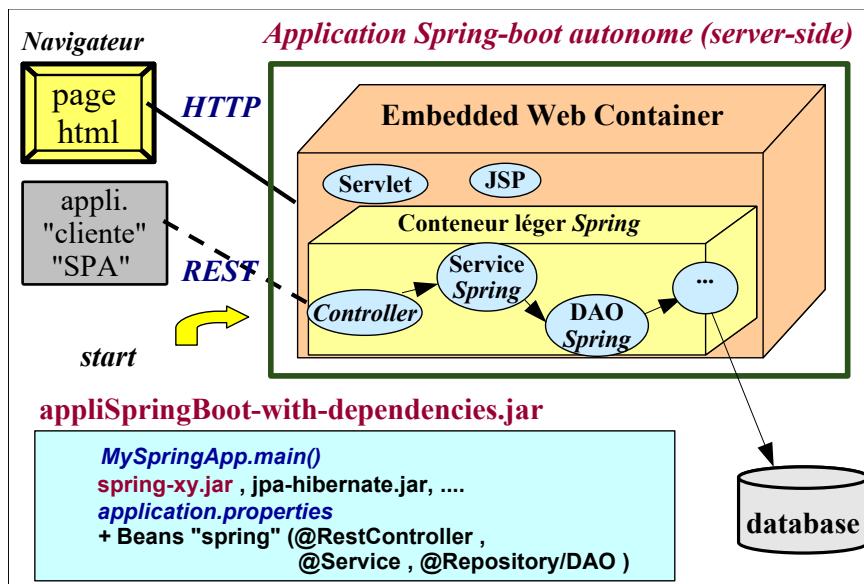
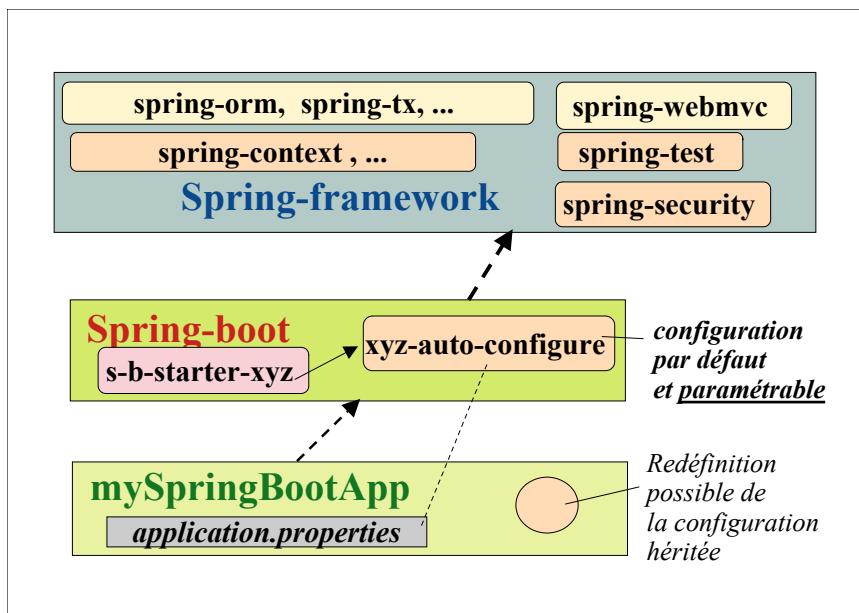
Cet exemple sera un peu compréhensible après vu le chapitre sur **@Transactional** .

III - Spring Boot (l'essentiel)

1. Fonctionnalités de SpringBoot

L'extension "spring-boot" permet (entre autre) de :

- démarrer une application java/web depuis un simple "main()" (sans avoir besoin d'effectuer un déploiement au sein d'un serveur de type de tomcat)
- simplifier la déclaration de certaines dépendances ("maven") via des héritages de configuration type (bonnes combinaisons de versions)
- (éventuellement) *auto-configurer une partie de l'application selon les librairies trouvées dans le classpath*.
- **Spring-boot** est assez souvent utilisé en coordination avec **Spring-MVC** (bien que ce ne soit pas obligatoire).



Quelques avantages d'une configuration "spring-boot" :

- **tests d'intégrations facilités** dès la phase de développement (l'application démarre toute seule depuis un main() ou un test JUnit sans serveur et l'on peut alors simplement tester le comportement web de l'application via selenium ou un équivalent).
- **déploiements simplifiés** (plus absolument besoin de préparer un serveur d'application JEE , de le paramétrier pour ensuite déployer l'application dedans).
- **Possibilité de générer un fichier ".war"** si l'on souhaite déployer l'application de façon standard dans un véritable serveur d'applications .
- **Configuration et démarrage très simples** (pas plus compliqué que node-js si l'on connaît bien java) .
- **Application java pouvant** (dans des cas simples) **être totalement autonome** si l'on s'appuie sur une base de données "embedded" (de type "H2" ou bien "HSQLDB").

Quelques traits particuliers (souvent perçus de façons subjectives) :

- Spring-boot (et Spring-mvc) sont des technologies propriétaires "Spring" qui s'écartent volontairement du standard officiel "JEE 6/7" pour se démarquer de la technologie concurrente EJB/CDI .
- Un web-service REST "java" codé avec Spring-boot + Spring-mvc comporte ainsi des annotations assez éloignées de la technologie concurrente CDI/Jax-RS bien qu'au final, les fonctionnalités apportées soient très semblables.

Attention (versions):

- Spring-boot 1.x compatible avec Spring 4.x
- Spring-boot 2.x compatible avec Spring 5.x (et utilisant beaucoup les nouveautés de java >=8) .

--> quelques différences (assez significatives) entre Spring-boot 1 et 2 .

2. Spring-Initializer



Spring Initializr
Bootstrap your application

"Spring Initializer" (<https://start.spring.io/>) est une application web en ligne disponible publiquement sur internet et qui permet de construire un point de départ d'une nouvelle application basée sur spring-boot .

Project	Language	
<input type="radio"/> Gradle - Groovy	<input checked="" type="radio"/> Java	<input type="radio"/> Kotlin
<input type="radio"/> Gradle - Kotlin	<input type="radio"/> Groovy	
<input checked="" type="radio"/> Maven		
Spring Boot		
<input type="radio"/> 3.2.2 (SNAPSHOT)	<input checked="" type="radio"/> 3.2.1	
<input type="radio"/> 3.1.8 (SNAPSHOT)	<input type="radio"/> 3.1.7	
Project Metadata		
Group	com.example	
Artifact	demo	
Name	demo	
Description	Demo project for Spring Boot	
Package name	com.example.demo	
Packaging	<input checked="" type="radio"/> Jar	<input type="radio"/> War
Java	<input type="radio"/> 21	<input checked="" type="radio"/> 17
<input style="border: 1px solid #ccc; padding: 5px; margin-right: 10px;" type="button" value="GENERATE"/> <input style="border: 1px solid #ccc; padding: 5px; margin-right: 10px;" type="button" value="EXPLORE"/> <input style="border: 1px solid #ccc; padding: 5px;" type="button" value="SHARE..."/>		

Dependencies

GraalVM Native Support DEVELOPER TOOLS

Support for compiling Spring applications to native executables using the GraalVM native-image compiler.

Lombok DEVELOPER TOOLS

Java annotation library which helps to reduce boilerplate code.

Spring Boot Dev Tools DEVELOPER TOOLS

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database SQL

génère **demo.zip** avec dans le sous répertoire "demo" ou autre (selon choix "artifactId") :
- pom.xml (avec les "starters" sélectionnés)

- **src/main/resources/application.properties**
- **src/main/java/.../.../DemoApplication** (avec main())
- **src/test/java/.../.../DemoApplicationTest** (avec JUnit)

NB : le contenu initial de pom.xml dépendra essentiellement de la sélection des technologies/starters .

demo.zip

Generate - Ctrl + G Explore - Ctrl + Space Share...

DOWNLOAD COPY

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi
3      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
4      <modelVersion>4.0.0</modelVersion>
5
6      <parent>
7          <groupId>org.springframework.boot</groupId>
8          <artifactId>spring-boot-starter-parent</artifactId>
9          <version>3.2.1</version>
10         <relativePath/> 
11     </parent>
12
13     <groupId>com.example</groupId>
14     <artifactId>demo</artifactId>
15     <version>0.0.1-SNAPSHOT</version>
16
17     <name>demo</name>
18     <description>Demo project for Spring Boot</description>
19
20     <properties>
21         <java.version>17</java.version>
22     </properties>
23
24     <dependencies>
25         <dependency>

```

3. Spring-boot (configuration et démarrage)

3.1. Exemple de démarrage avec Spring-Boot

```
package tp;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

//NB: @SpringBootApplication est un équivalent
// de @Configuration + @EnableAutoConfiguration + @ComponentScan/current package

@SpringBootApplication
public class MySpringBootApplication {

    public static void main(String[] args) {
        SpringApplication.run(MySpringBootApplication.class, args);
        System.out.println("http://localhost:8080/myMvcSpringBootApp");
    }
}
```

==> la partie **@EnableAutoConfiguration** de **@SpringBootApplication** fait que le fichier **application.properties** sera automatiquement analysé .

==> il faut absolument que les classes de tests et de configuration (ex : **tp.config.WebSecurityConfig**) soient placées dans des sous-packages car le **@ComponentScan** de **@SpringBootApplication** est par défaut configuré pour n'analyser que le package courant (ici **tp**) et ses sous packages .

```
package tp.test;

import org.junit.Assert; import org.junit.Test;
import org.slf4j.Logger; import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import tp.MySpringBootApplication;

@SpringBootTest(classes= {MySpringBootApplication.class})
public class TestServiceXy {
    private static Logger logger = LoggerFactory.getLogger(TestServiceXy.class);

    @Autowired
    private ServiceXy service; // service métier à tester

    @Test
    public void testQuiVaBien() {
        logger.debug("testQuiVaBien");
        Assert.assertTrue(1+1==2);
    }
}
```

3.2. Configuration maven pour spring-boot 3 (et spring 6)

```

...
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.1</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<properties>
    <java.version>17</java.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <!-- <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency> -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <!-- spring-boot-devtools useful for refresh without restarting -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>

<build>
    <finalName>${project.artifactId}</finalName>
</build>
...

```

3.3. Rare boot (standalone) sans annotation

```
ConfigurableApplicationContext context =
    SpringApplication.run(MyApplicationConfig.class);
ServiceXy serviceXy = context.getBean(ServiceXy.class);
....
context.close() ;
```

ou bien (en plusieurs phases mieux contrôlées) :

```
SpringApplication app = new SpringApplication(DomainAndPersistenceConfig.class);
app.setLogStartupInfo(false);
ConfigurableApplicationContext context = app.run(args);
```

Sans l'annotation `@SpringBootApplication` sur la classe de démarrage , la configuration (`@ComponentScan` , ...) doit être explicitée sur la classe de configuration passée en argument du constructeur (ou bien de la méthode `run()`).

3.4. Boot (standalone) avec annotation `@SpringBootApplication`

Rappel du démarrage avec `@SpringBootApplication`

```
package tp;

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

//NB: @SpringBootApplication est un équivalent
// de @Configuration + @EnableAutoConfiguration + @ComponentScan/current package

@SpringBootApplication
public class MySpringBootApplication {

    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(MySpringBootApplication.class);
        // app.setAdditionalProfiles("p1","p2","p3");
        ConfigurableApplicationContext context = app.run(args);
        System.out.println("http://localhost:8080/springApp");
    }
}
```

3.5. Structure minimaliste d'une application SpringBoot :

NB : avec spring-boot et un packaging "jar" (et pas "war") , le répertoire src/main/webapp n'existe pas et il faut alors placer les ressources web (.html , .css , ...) dans le sous répertoire "static" (à éventuellement créer) de src/main/resources .

springApp

```
pom.xml
src/main/java
    tp.appliSpring.MySpringBootApplication.java
    tp.appliSpring.entity
    tp.appliSpring.dao ou tp.appliSpring.repository
    tp.appliSpring.service
    ...
src/main/resources
    application.properties
    static
        index.html
src/test/java
    tp.appliSpring.TestApp
```

application.properties

```
server.servlet.context-path=/springApp
server.port=8080
```

static/index.html

```
<html>
    <body>
        <h1>Hello world , springApp</h1>
    </body>
</html>
```

NB :

- un début de structure est préparé par spring-initializer
- il faut idéalement compléter la partie static/index.html et application.properties de manière à effectuer un premier lancement et tester l'application avec un navigateur internet (ex : <http://localhost:8080/springApp>)

3.6. Tests unitaires avec Spring-boot

éventuellement :

```
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

//{@RunWith(SpringJUnit4ClassRunner.class)
@ExtendWith(SpringExtension.class) //si junit5/jupiter
@SpringApplicationConfiguration(classes = MyApplicationConfig.class)
//au lieu du classique @ContextConfiguration(....)
public class MyApplicationTest {
...
}
```

ou mieux encore :

```
package tp.appliSpring.service;
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.slf4j.Logger; import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import tp.MySpringBootApplication;

// @RunWith(SpringRunner.class) si junit4
@ExtendWith(SpringExtension.class) // junit5/jupiter
@SpringBootTest(classes= {MySpringBootApplication.class})
public class TestServiceXy {
    private static Logger logger = LoggerFactory.getLogger(TestServiceXy.class);

    @Autowired
    private ServiceXy service; // service métier à tester

    @Test
    public void testQuiVaBien() {
        logger.debug("testQuiVaBien");
        assertTrue(1+1==2);
    }
}
```

NB : **@SpringBootTest()** sans aucune autre indication active une configuration automatique intelligente qui va déterminer/détecter :

- la classe principale de l'application SpringBoot (celle qui est annotée par **@SpringbootApplication** et qui fait office de configuration principale par défaut).
- S'il faut utiliser **@RunWith(SpringRunner.class)** ou bien **@ExtendWith(SpringExtension.class)** en fonction de ce qui est présent dans le classpath (paramétré via pom.xml).

3.7. auto-configuration (facultative mais conseillée)

L' annotation **@EnableAutoConfiguration** (à placer à coté du classique **@Configuration** de **java-config** et déjà intégrée dans **@SpringBootApplication**) demande à **Spring Boot** via la classe **SpringApplication** de **configurer automatiquement l'application en fonction des bibliothèques trouvées dans son class-path** (indirectement défini via le contenu de pom.xml) et en fonction de **application.properties** .

Par exemple:

- Parce que les bibliothèques Hibernate sont dans le Classpath, le bean *EntityManagerFactory* de JPA sera implémenté avec Hibernate.
- Parce que la bibliothèque du SGBD H2 est dans le Classpath, le bean "dataSource" sera implémenté avec H2 (avec administrateur par défaut "sa" et sans mot de passe) .

Le "dialecte" hibernate sera également auto-configuration pour "H2" .

Cette auto-configuration ne fonctionne qu'avec des bases "embedded" (H2 , hsqldb, ...)
Pour les autres bases (mysql, mariadb, postgres, oracle, db2, ...) une configuration complémentaire est nécessaire dans application.properties .

- Parce que la bibliothèque [spring-tx] est dans le Classpath, c'est le gestionnaire de transactions de Spring qui sera utilisé.
- Parce que une bibliothèque "spring...security" sera trouvée dans le classpath , l'application java/web sera automatiquement sécurisée (en mode basic-http) avec un username "user" et un mot de passe qui s'affichera au démarrage de l'application dans la console .
- ...

3.8. Configuration des logs avec springBoot

NB: par défaut, **spring-boot** utilise pour l'instant "**slf4j**"+"**log4j 2**" par défaut pour générer des lignes de log.

Paramétrage des logs dans application.properties:

```
logging.level.root=INFO
logging.level.org.springframework=ERROR
logging.level.org.mycontrib=TRACE
#logging.level.org.springframework.security=DEBUG
```

3.9. Auto-configuration "spring-boot" avec application.properties

Rappel : l'annotation **@SpringBootApplication** (*placée sur la classe de démarrage*)

est un équivalent de

@Configuration + @EnableAutoConfiguration + @ComponentScan/current package

Dans certains cas (classiques, simples), la configuration de l'application spring-boot peut entièrement être placée dans le fichier **application.properties** (de scr/main/resources) .

Le fichier **application.properties** est implicitement analysé en mode **@EnableAutoConfiguration** et peut comporter tous un tas de propriétés (dont les noms sont normalisés dans la documentation de référence de spring) .

Beaucoup de propriétés de **application.properties** peuvent considérées comme une alternative hyper simplifiée d'un énorme paquet de configuration explicite (xml ou java) qui était auparavant placé dans une multitude de fichiers complémentaires (ex : WEB-INF/web.xml , META-INF/persistence.xml , ... ou XyJavaConfig.class) .

Exemple de fichier **application.properties**

```
server.servlet.context-path=/myMvcSpringBootApp
server.port=8080
logging.level.org=INFO

spring.mvc.view.prefix=/views/
spring.mvc.view.suffix=.jsp

#spring.datasource.driverClassName=com.mysql.jdbc.Driver
#spring.datasource.url=jdbc:mysql://localhost:3306/mydb
#spring.datasource.username=root
#spring.datasource.password=

spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:~/mydb
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.platform=h2
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.hibernate.ddl-auto=create
#enable spring-data (generated dao implementation classes)
spring.data.jpa.repositories.enabled=true
```

Attention (gros piège) :

Le fichier **application.properties** est placé dans **src/main/resources** et est analysé/traité comme une ressource par maven . Par défaut, certaines versions de maven ne supportent pas des caractères exotiques (qui ne sont pas UTF-8) dans les ressources. **Il ne faut donc surtout pas placer des caractères accentués (é, à , è , ..) mal encodés (pas UTF-8) dans le fichier application.properties sinon ça bloque tout et les message d'erreurs ne sont pas parlants/explicites .**

3.10. Profiles 'spring" (variantes dans les configurations)

NB : Les profiles "spring" (variantes de configurations) peuvent éventuellement être complémentaires . L'annotation **@Profile()** peut être placée sur un composant Spring ordinaire (préfixé par exemple par **@Component**) ou bien sur une classe de configuration (**@Configuration**) .

Exemple :

```
import javax.annotation.PostConstruct;
import org.springframework.context.annotation.Profile;
...
@Component
@Profile("reInit")
public class ReInitDefaultDataSet {
    @Autowired
    private DeviseService deviseService;

    @PostConstruct
    public void initDataSet() {
        deviseService.saveOrUpdate(new Devise("EUR","Euro",1.0));
        deviseService.saveOrUpdate(new Devise("USD","Dollar",1.1243));
    }
}
```

Ce composant (servant ici à initialiser un jeu de données en base) ne sera activé et utilisé au sein de l'application Spring que si le profile "reInit" est activé .

D'autre part, **le framework "spring" analyse automatiquement les fichiers application-profileName.properties (en complément de application.properties) si le profile "profileName" est activé au démarrage de l'application.**

NB: Si un même paramètre a des valeurs différentes dans application.properties et application-profileName.properties , la valeur retenue sera celle du profile activée .

Exemple :

application-reInit.properties

```
#database tables will be dropped & re-created at each new restart of the application or tests
# (dev only) ,CREATE TABLE will be generated from @Entity structure
spring.jpa.hibernate.ddl-auto=create
```

Activation explicite d'un profile "spring" au démarrage d'une application :

```
...
@SpringBootApplication
public class MySpringBootApplication extends SpringBootServletInitializer {
    public static void main(String[] args) {
        //SpringApplication.run(MySpringBootApplication.class, args);
        SpringApplication app = new SpringApplication(MySpringBootApplication.class);
        app.setAdditionalProfiles("embeddedDb","reInit","appDbSecurity");
        ConfigurableApplicationContext context = app.run(args);
        //securité par défaut si la classe WebSecurityConfig n'existe pas dans l'application:
        //System.out.println("default username=user et password précisé au démarrage");
    }
}
```

Activation automatique d'un profile "spring" via des propriétés d'environnement :

```
java .... -Dspring.profiles.active=reInit,embeddedDb
```

Activation d'un profile "spring" au démarrage d'un test unitaire :

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes= {MySpringBootApplication.class})
@ActiveProfiles("reInit,embeddedDb,permitAllSecurity")
public class TestXy {
    ...
}
```

Astuce avec docker :

```
docker container run -p 8181:8181 -e SPRING_PROFILES_ACTIVE=withSecurity,oracle -d ....
```

L'option **-e** de "docker container run" permet de fixer la valeur d'une variable d'environnement au moment du lancement/démarrage du conteneur docker. Et la variable d'environnement **SPRING_PROFILES_ACTIVE** (comme la propriété système `-Dspring.profiles.active`) est fondamentale pour paramétrer la liste des profils spring à activer au démarrage de l'application.

IV - Spring AOP (intercepteurs, aspects)

1. Spring AOP (essentiel)

1.1. Technologies AOP et "Spring AOP"

- ◆ AOP (Aspect Oriented Programming) est un complément à la programmation orientée objet.
- ◆ AOP consiste à programmer une bonne fois pour toute certains aspects techniques (logs , sécurité , transaction, ...) au sein de classes spéciales.
- ◆ Une configuration (xml ou ...) permettra ensuite à un framework AOP (ex: AspectJ ou Spring-AOP) d'appliquer (par ajout automatique de code) ces aspects à certaines méthodes de certaines classes "fonctionnelles" du code de l'application.
- ◆ Vocabulaire AOP:
 - PointCut* : endroit du code (fonctionnel) où seront ajoutés des aspects
 - Advice* : ajout de code/aspect (avant, après ou bien autour de l'exécution d'une méthode)
- ◆ On parle de tissage ("weaver") du code :
Le code complet est obtenu en tissant les fils/aspects techniques avec les fils/méthodes fonctionnel(le)s .

Les mécanismes de Spring AOP (en version $\geq 2.x$) sont toujours dynamiques (déclenchés lors de l'exécution du programme) . Spring AOP 2 utilise néanmoins des syntaxes de paramétrage (annotations) volontairement proches du standard de fait java "AspectJ-weaver" .

Dans pom.xml

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
</dependency> <!-- et indirectement spring-bean, spring-core , spring-aop-->

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>${spring.version}</version>
</dependency> <!-- et indirectement aspectj-weaver-->
```

1.2. Mise en oeuvre rapide de Spring aop via des annotations

```
package util;
//Nécessite quelquefois aspectjrt.jar , aspectjweaver.jar
//(de spring.../lib/aspectj)
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
@Component
public class MyPerfLogAspect {

    @Around("execution(* xxx.services.*.*(..))")
    public Object doXxxLog(ProceedingJoinPoint pjp)
    throws Throwable {
        System.out.println("<< trace == debut == "
            + pjp.getSignature().toLongString() + " <<");
        long td=System.nanoTime();
        Object objRes = pjp.proceed();
        long tf=System.nanoTime();
        System.out.println(">> trace == fin == "
            + pjp.getSignature().toShortString() +
            " [ " + (tf-td)/1000000.0 + " ms] >>");
        return objRes;
    }
}
```

avec **@Around("execution(typeRetour package.Classes.methode(..))")**

et dans **<aop:aspectj-autoproxy/>** dans une configuration spring xml
ou bien **@EnableAspectJAutoProxy** sur une classe de **@Configuration** en mode java-config .

Ancienne configuration aop en pur xml (sans annotations dans la classe java de l'aspect)

```

...
<bean id="myLogAspectBean" class="tp...MyPerfLogAspect"></bean>
<aop:config>
    <aop:pointcut id="execution_methodes_package_livre"
        expression="execution(* tp.bibliotheque.livres.*.*.*(..))" />

    <aop:pointcut id="execution_methodes_package_ab_emp"
        expression="execution(* tp.bibliotheque.ab_emp.*.*.*(..))" />

    <aop:aspect id="myLogAspect" ref="myLogAspectBean" >
        <aop:around method="doXxxLog"
            pointcut-ref="execution_methodes_package_livre" />
        <aop:around method="doXxxLog"
            pointcut-ref="execution_methodes_package_ab_emp" />
    </aop:aspect>
</aop:config>
```

1.3. Types d'advice

Type d'advice	significations
<code>@Before("pointcut_expression")</code> <code>beforeXyzAdvice(JointPoint jp)</code>	Avant l'exécution du code
<code>@AfterReturning(pointcut ="pointcut_expression" ,</code> <code>returning = "res")</code> <code>afterReturningXyzAdvice(JoinPoint jp , Object res)</code>	Après l'exécution sans exception du code (en ayant accès à la valeur de retour)
<code>@AfterThrowing(pointcut = "pointcut_expression",</code> <code>throwing = "ex")</code> <code>afterThrowingxyzAdvice(JoinPoint jp , Exception ex)</code>	Après l'exécution avec exception du code (en ayant accès à l'exception)
<code>@After("pointcut_expression")</code> <code>afterXyzAdvice(JointPoint jp)</code>	Après l'exécution du code (sans détails)
<code>@Around("pointcut_expression")</code> <code>aroundXyzAdvice(ProceedingJointPoint pjp)</code>	Autour de l'exécution du code

soit à peu cette logique là :

```

try {
    //@Before
    method();
    //@AfterReturning
} catch(Throwable t) {
    //@AfterThrowing
} finally {
```

```
//@After
}
```

Exemple en version plus élaborée :

```
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
...

@Aspect
@Component
@Profile("perf") //penser à @EnableAspectJAutoProxy sur une des classes de config
// et à System.setProperty("spring.profiles.active", "perf"); ou @ActiveProfiles({ "perf" }) sur Test
public class MyLoggingAspect {

@Pointcut("execution(* tp.appliSpring.core.service.*.*(..))")
public void servicePointcut(){
}

@Pointcut("execution(* tp.appliSpring.core.dao.*.*(..))")
public void daoPointcut(){
}

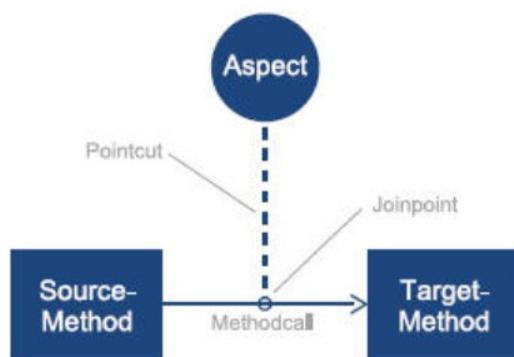
@AfterThrowing (pointcut = "servicePointcut()", throwing = "ex" )
public void logAfterThrowingAllServiceMethodsAdvice(JoinPoint jp , Exception ex ) throws Throwable
{
    System.out.println("**** LoggingAspect" + ex + " in " + jp.getSignature().toLongString());
}

@Before ("servicePointcut()")
public void beforeServiceMethodsAdvice(JoinPoint jp) throws Throwable
{
    System.out.print("**** beforeServiceMethodsAdviceLoggingAspect: "
        +jp.getSignature().toLongString() + "was called with args= ");
    for(Object a : jp.getArgs()) { System.out.print(" " + a ); }
    System.out.print("\n");
}

@AfterReturning (pointcut = "servicePointcut()", returning = "res")
public void logAfterReturningServiceMethodsAdvice(JoinPoint jp , Object res) throws Throwable
{
    System.out.println("**** logAfterReturningServiceMethodsAdvice : returned value = " + res
    + " after execution of " +jp.getSignature().toLongString());
}

/*
* ProceedingJoinPoint is an extension of JoinPoint with .proceed() additional method()
}
```

```
/*
 *
 * @Around("servicePointcut() || daoPointcut()")
 * public Object doPerfLogAdvice(ProceedingJoinPoint pjp) throws Throwable {
 *     System.out.println("<< trace == debut == " + pjp.getSignature().toLongString() + " <<");
 *     long td = System.nanoTime();
 *     Object objRes = pjp.proceed();
 *     //Object objRes = pjp.proceed(pjp.getArgs());
 *     long tf = System.nanoTime();
 *     System.out.println(">> trace == fin == " + pjp.getSignature().toShortString()
 *                     + " [" + (tf - td) / 1000000.0 + " ms] >>");
 *     return objRes;
 * }
```



Autres exemples de définitions de "pointcut" :

```
//@within pour annotation @Aff (avec @Target(ElementType.TYPE) )
//placée sur l'ensemble d'une classe
@Pointcut("@within(tp.appliSpring.annotation.Aff)")
public void annotAffPointcut(){
}
```

```
//@annotation pour annotation @LogExecutionTime (avec @Target(ElementType.METHOD))
// placée sur une méthode (sujet de l'application de l'aspect)
@Pointcut("@annotation(tp.appliSpring.annotation.LogExecutionTime)")
public void annotLogExecutionTimePointcut(){
}
```

```
@Around("exemplePointcut() && annotAffPointcut()")
//@Around("annotLogExecutionTimePointcut()")
public Object doXxxLog(ProceedingJoinPoint pip) throws Throwable { .... }
```

1.4. Spring-AOP à l'ancienne (avec Advisor/intercepteur , proxies)

```
public interface Advisor {
    ...
    Advice getAdvice();
}

public interface IntroductionAdvisor extends Advisor{
    ...
}

public class DefaultIntroductionAdvisor implements IntroductionAdvisor{
    ....
}
```

interface basique **Thing**

```
public interface Thing {
    String getName();
    void setName(String name);

    String getValue();
    void setValue(String value);

    String toString();
}
```

implémentation basique **BasicThing**

```
public class BasicThing implements Thing {
    private String name;
    private String value;
    ...
}
```

advice/intercepteur "**MyLoggingAdvice**"

```
...
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class MyLoggingAdvice implements MethodInterceptor {

    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("** Method name : " + invocation.getMethod().getName());
        System.out.println("** Method arguments : " +
                           Arrays.toString(invocation.getArguments()));
        Object result = invocation.proceed();
        System.out.println("** Method result : " + result.toString());
        return result;
    }
}
```

Cet advice aop servira à générer des lignes de log sans altérer la valeur de retour

advice/intercepteur "*UppercaseInterceptor*"

```
public class UppercaseInterceptor implements MethodInterceptor {
    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        Object result = invocation.proceed();
        if(result instanceof String) {
            result=((String) result).toUpperCase();
        }
        return result;
    }
}
```

Cet intercepteur aop servira à transformer le résultat en majuscules lorsqu'il est de type "String".

```
import org.springframework.aop.framework.ProxyFactoryBean;
import org.springframework.aop.support.DefaultIntroductionAdvisor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyOldAopConfig {

    @Bean
    Thing basicThing() { return new BasicThing("nb_continents" , "5"); }

    @Bean
    Thing basicThingProxy(Thing thing ) {
        ProxyFactoryBean proxyFactoryBean = new ProxyFactoryBean();
        proxyFactoryBean.setTarget(thing);
        //proxyFactoryBean.setInterceptorsNames("myLoggingAdvisor"); //name resolution failed
        //proxyFactoryBean.addAdvice(new MyLoggingAdvice()); //ok
        proxyFactoryBean.addAdvisor(
            new DefaultIntroductionAdvisor(new MyLoggingAdvice()));//ok
        proxyFactoryBean.addAdvisor(
            new DefaultIntroductionAdvisor(new UppercaseInterceptor()));//ok
        proxyFactoryBean.setInterfaces(Thing.class);
        return (Thing) proxyFactoryBean.getObject();
    }
}
```

Au sein de cette configuration importante, la configuration AOP n'est pas automatique mais est ici hyper explicite (à l'ancienne comme durant le début des années 2000).

La classe **ProxyFactoryBean** permet la **création dynamique d'un proxy qui va appliquer une succession d'intercepteurs avant d'invoquer les méthodes ordinaires d'un bean de base** (implémentation basique) d'une manière ressemblante au design pattern "décorateur".

Ce proxy/bean dynamique aura la même interface que le bean de base et s'utilisera de la même manière. Cependant les méthodes invoquées seront automatiquement enrichies par les intercepteurs enregistrés.

Utilisation :

```
...
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SimpleAopTestApp {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext springContext = new
            AnnotationConfigApplicationContext(MyOldAopConfig.class);
        //Thing chose = (Thing) springContext.getBean("basicThing");
        Thing chose = (Thing) springContext.getBean("basicThingProxy");
        System.out.println(chose.getName() + ":" + chose.getValue());
        System.out.println("chose=" + chose.toString());
        springContext.close();
    }
}
```

Résultats :

```
** Method name : getName
** Method arguments : []
** Method result : NB_CONTINENTS
** Method name : getValue
** Method arguments : []
** Method result : 5
NB_CONTINENTS:5
** Method name : toString
** Method arguments : []
** Method result : BASICTHING [NAME=NB_CONTINENTS, VALUE=5]
chose=BASICTHING [NAME=NB_CONTINENTS, VALUE=5]
```

Ancienne configuration XML (dès années 2005-2015) :

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="basicThing" class="tp.appliSpring.exemple_advisor.BasicThing">
        <property name="name" value="nb_continents" />
        <property name="value" value="5" />
    </bean>

    <bean id="myLoggingAdvisor" class="...." />
    <bean id="uppercaseInterceptor" class="...." />

    <bean id="basicThingProxy"
          class="org.springframework.aop.framework.ProxyFactoryBean">

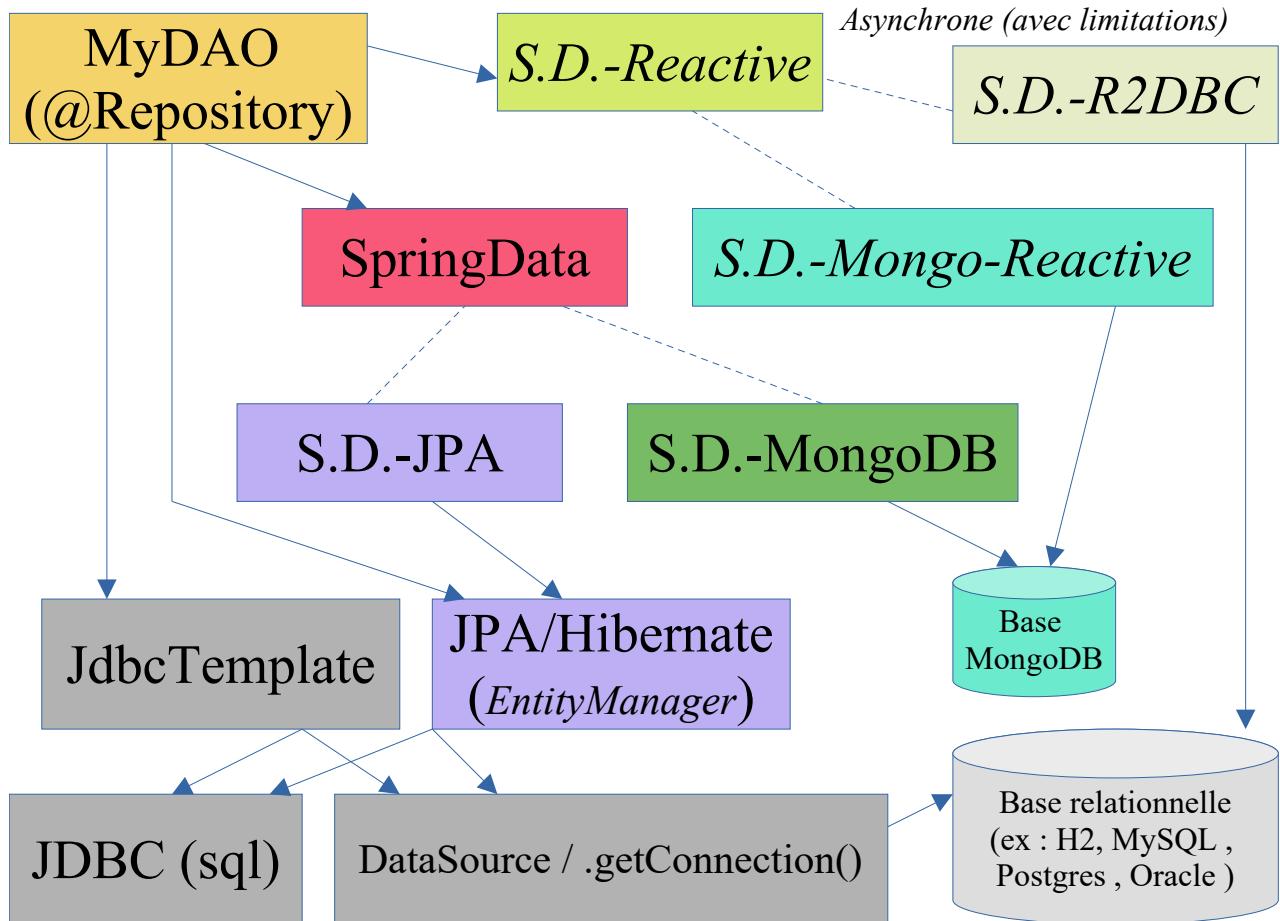
        <property name="target" ref="basicThing" />

        <property name="interceptorNames">
            <list><value>myLoggingAdvisor</value>
                <value>uppercaseInterceptor</value>
            </list>
        </property>
    </bean>
</beans>
```

V - Accès aux données depuis Spring (jdbc,JPA)

1. Accès au données via Spring (possibilités)

Accès aux données via Spring (principales possibilités)



2. Utilisation de Spring au niveau des services métiers

2.1. Dépendances classiques



Business **@Service** ---> Data Access Object (**@Repository**) ---> javax.sql.DataSource

Principales variantes au niveau du DAO:

- JDBC seulement
- JPA/Hibernate
- SpringData et base SQL ou NoSQL

2.2. Principales fonctionnalités d'un service métier

- Contrôler / superviser une séquence de traitements élémentaires sur quelques entités.
- Offrir des méthodes «créerXx rechercherXx , majXx , supprimerXx» (C.R.U.D.) dont le code interne consistera essentiellement à déléguer ces opérations de persistance aux D.A.O. (génériques ou spécifiques).
- Comporter des règles de gestions (méthodes vérifierXxx() , vérifierYyy()).
- Offrir des méthodes spécifiques à l'objet métier considéré (ex: transferer() ,)
- Gérer/superviser des transactions (commit / rollback).

2.3. Vision abstraite d'un service métier

Interface abstraite avec méthodes *métiers* ayant:

- des POJOs de données en paramètres d'entrée et/ou en sortie (valeur de retour)
- des remontées d'exceptions métiers uniformes (héritant de *Exception* ou bien *RuntimeException*) quelque soit la technologie utilisée en arrière plan.

exemple:

```
public class MyApplicationException extends RuntimeException {  
//public class MyApplicationException extends Exception {  
  
    private static final long serialVersionUID = 1L;  
  
    public MyApplicationException() { super();}  
    public MyApplicationException(String msg) {super(msg); }  
    public MyApplicationException(String msg,Throwable cause) {super(msg,cause); }  
}
```

et

```
public interface ServiceCompte {  
  
    public Compte getCompteByNum(long numCpt) throws MyApplicationException;  
    ...  
    public void transferer(long numCompteADebiter,  
                          long numCompteACrediter,  
                          double montant) throws MyApplicationException;  
}
```

3. DataSource JDBC (vue Spring)

3.1. DataSource élémentaire (sans pool)

@Configuration

```
public class DataSourceConfig {  
  
    @Bean(name="datasource") //by default beanName is same of method name  
    public DataSource dataSource() {  
        DriverManagerDataSource dataSource = new DriverManagerDataSource();  
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");  
        dataSource.setUrl("jdbc:mysql://localhost:3306/minibank_db_ex1");  
        dataSource.setUsername("root");  
        dataSource.setPassword("root");//"root" ou "formation" ou ..."."  
        return dataSource;  
    }  
}
```

Remarques:

La classe "**org.springframework.jdbc.datasource.DriverManagerDataSource**" est une version basique (sans pool de connexions recyclables , juste pour les tests) et qui a l'avantage de ne pas nécessiter de ".jar" supplémentaire.

Seules choses à bien mettre en place (dans le ClassPath) :

- le ".jar" contenant le code du **driver JDBC** pour "MySql" ou "Oracle" ou "..." (ex: *mysql-connector-java-.....jar*)
- spring-jdbc (directement ou indirectement)

NB : Dans un contexte "spring-boot" + "@EnableAutoconfiguration" , il suffit de paramétrier le fichier de configuration principal **application.properties** :

```
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.url=jdbc:h2:~/mydb  
spring.datasource.username=sa  
spring.datasource.password=  
spring.datasource.platform=h2
```

3.2. Embedded DataSource with pool

De façon à avoir de meilleures performances en mode "production" , on pourra utiliser des implémentations plus sophistiquées d'un dataSource jdbc embarqué dans l'application (spring-boot ou autre) :

La classe "***org.apache.commons.dbcp.BasicDataSource***" (de la librairie "**common-dbcp**" de la communauté Apache) correspond à une technologie que l'on peut intégrer facilement un peu partout (dans une application autonome , dans une application web (.war) ,).

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.1</version>
</dependency>
```

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
```

```
BasicDataSource dataSource = org.apache.commons.dbcp2.BasicDataSource() ;
dataSource.setUrl(...); ...
```

La technologie alternative **c3p0** (souvent utilisée avec hibernate) est également une bonne mise en oeuvre de "embedded jdbc dataSource with pool" .

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-c3p0</artifactId>
    <version>5.4.10.Final</version>
</dependency>
```

et paramétrages avancés de ce type :

```
... c3p0.min_size=5
... c3p0.max_size=20
... c3p0.acquire_increment=5
... c3p0.timeout=1800
```

Alternative (encore plus moderne/performante et utilisée par défaut par SpringBoot) : **HirakiCP**

```
spring.datasource.hikari.connectionTimeout=30000
spring.datasource.hikari.idleTimeout=600000
spring.datasource.hikari.maxLifetime=1800000
```

```
HikariConfig config = new HikariConfig();
config.setJdbcUrl( "jdbc_url" );
config.setUsername( "database_username" );
config.setPassword( "database_password" );
config.addDataSourceProperty( "cachePrepStmts" , "true" );
config.addDataSourceProperty( "prepStmtCacheSize" , "250" );
config.addDataSourceProperty( "prepStmtCacheSqlLimit" , "2048" );
HikariDataSource ds = new HikariDataSource( config );
```

4. DAO Spring basé directement sur JDBC

Pour les cas simples , on peut s'appuyer directement sur JDBC (*sans utiliser JPA/Hibernate ni Spring-Data*) .

4.1. Avec **JdbcDaoSupport**

De façon à coder rapidement une classe d'implémentation concrète d'un DAO basée directement sur la technologie JDBC on pourra avantageusement s'appuyer sur la classe abstraite **JdbcDaoSupport**.

exemple:

```
public class JdbcXxxDAO extends JdbcDaoSupport implements XxxDAO
{
    @Autowired
    @Override
    public void setDataSource(DataSource ds){
        super.setDataSource(ds);
    }
    ...
}
```

Principales fonctionnalités héritées de **JdbcDaoSupport**:

==> **getDataSource()** permet de récupérer au niveau du code la source de données JDBC qui a été obligatoirement injectée.

==> **setDataSource(DataSource ds)** permet d'injecter la source de données JDBC .

==> **getConnection()** , **releaseConnection(cn)** permet d'obtenir et libérer une connexion JDBC .

NB: Prises en charge par Spring , les méthodes **JdbcDaoSupport.getConnection()** et **JdbcDaoSupport.releaseConnection()** seront automatiquement synchronisées avec le contexte transactionnel du thread courant (cn.close() différé après la fin de la transaction,).

==> **getJdbcTemplate()** permet de récupérer un objet de plus haut niveau (de type **JdbcTemplate**) qui libère automatiquement la connexion en fin d'opération et qui permet de simplifier un peu la syntaxe.

.../...

Exemple de code (rare) n'utilisant pas de "JdbcTemplate":

```
public class JdbcInfosAccesDAO extends JdbcDaoSupport implements InfosAccesDAO {  
    ...  
    public InfosAcces getVerifiedInfosAccesV1(String userName, String password)  
        throws DataAccessException {  
        InfosAcces infos=null;  
        Connection cn = null;  
        try  
        {  
            cn = this.getConnection();  
            PreparedStatement prst = cn.prepareStatement("select NUM_CLIENT FROM  
                INFOSACCES WHERE username=? and password=?");  
            prst.setString(1,userName);  
            prst.setString(2,password);  
            ResultSet rs = prst.executeQuery();  
            if(rs.next())  
            {  
                infos=new InfosAcces();  
                infos.setUserName(userName); infos.setPassword(password);  
                infos.setClient_id(rs.getLong("NUM_CLIENT"));  
            }  
            rs.close();  
            prst.close();  
        }  
        catch(SQLException se)  
        {  
            se.printStackTrace();  
            throw new DataRetrievalFailureException(se.getMessage());  
        }  
        finally  
        {  
            this.releaseConnection(cn);  
        }  
        return infos;  
    }  
    ....  
}
```

4.2. Avec JdbcTemplate

NB : `JdbcTemplate` est créé à partir de `dataSource` (via `.getJdbcTemplate()` de `JdbcDaoSupport`)

Un objet de type `JdbcTemplate` :

- simplifie beaucoup l'api JDBC (`Statement` , `ResultSet`, ...)
- obtient et libère la connexion JDBC automatiquement depuis le `dataSource`
- collabore bien avec la logique `@Transactional` au niveau des `@Service`

Exemple de code :

```
package org.mycontrib.api.dao.jdbc;

import java.util.Map;

import javax.annotation.PostConstruct;
import javax.sql.DataSource;

...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.support.JdbcDaoSupport;
import org.springframework.stereotype.Repository;

@Repository
public class NewsDaoJdbc extends JdbcDaoSupport implements NewsDao {

    @Autowired
    private DataSource appDataSource;

    @PostConstruct
    private void initialize() {
        setDataSource(appDataSource);
    }

    @Override
    public News findNewsById(Long id) {
        News news=null;
        JdbcTemplate jt = this.getJdbcTemplate();

        Map map= jt.queryForMap(
            "SELECT id_news,text FROM News WHERE id_news=?",
            id);
        if(map != null && map.size() > 0)
            { news=new News((Long)map.get("id_news"),
                           (String)map.get("text"));
            }
        return news;
    }

    @Override
    public News insertNews(News n) {
        JdbcTemplate jt = this.getJdbcTemplate();
```

```

jt.update("INSERT INTO News(id_news,text) VALUES (?,?)",
          n.getId_news(), n.getText());
return n;
}

@Override
public News updateNews(News n) {
    JdbcTemplate jt = this.getJdbcTemplate();
    jt.update("UPDATE News SET text=? WHERE id_news=?",
              n.getText(),n.getId_news());
    return n;
}

```

Si nécessaire (selon contexte) :

```

import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.PlatformTransactionManager;

```

@Configuration

```

public class TxForJdbcTemplateConfig {
    @Bean //default name = method name = "jdbcTxManager"
    public PlatformTransactionManager jdbcTxManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}

```

@Service

```

@Transactional(transactionManager = "jdbcTxManager")
public class ServiceNewsImpl implements ServiceNews {
    @Autowired
    private NewsDao newsDao;

    @Override
    public News addNews(News n) {
        return newsDao.insertNews(n);
    }
}

```

4.3. Avec NamedParameterJdbcTemplate et RowMapper

```
@Configuration
public class DataSourceConfig {
//...

//utile pour le dao en version Jdbc (avec NamedParameterJdbcTemplate):
@Bean()
public NamedParameterJdbcTemplate namedParameterJdbcTemplate( DataSource
    dataSource) {
    return new NamedParameterJdbcTemplate(dataSource);
}
}
```

DaoCompteJdbc.java (code partiel)

```
...
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;

@Repository // @Component de type DAO/Repository
@Qualifier("jdbc")
public class DaoCompteJdbc /*extends JdbcDaoSupport*/ implements DaoCompte {

    private final String INSERT_SQL = "INSERT INTO compte(label, solde) values(:label,:solde)";
    private final String UPDATE_SQL =
        "UPDATE compte set label=:label , solde=:solde where numero=:numero";
    private final String FETCH_ALL_SQL = "select * from compte";
    private final String FETCH_BY_NUM_SQL = "select * from compte where numero=:numero";
    private final String DELETE_BY_NUM_SQL = "delete from compte where numero=:numero";

    @Autowired
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    @Override
    public List<Compte> findAll() {
        return namedParameterJdbcTemplate.query(FETCH_ALL_SQL, new CompteMapper());
    }

    @Override
    public void deleteById(Long numCpt) {
        SqlParameterSource parameters = new MapSqlParameterSource()
            .addValue("numero", numCpt);
        namedParameterJdbcTemplate.update(DELETE_BY_NUM_SQL, parameters);
    }

    @Override
```

```

public Compte findById(Long numCpt) {
    Compte compte = null;
    Map<String, Long> parameters = new HashMap<String, Long>();
    parameters.put("numero", numCpt);
    /*
     try {
        compte = namedParameterJdbcTemplate.queryForObject(FETCH_BY_NUM_SQL,
            parameters,
            new CompteMapper());
    } catch (DataAccessException e) {
        //e.printStackTrace();
        System.err.println(e.getMessage());
    }
    */
    List<Compte> comptes = namedParameterJdbcTemplate.query(FETCH_BY_NUM_SQL,
        parameters, new CompteMapper());
    compte = comptes.isEmpty()?null:comptes.get(0);
    return compte;
}

public Compte insert(Compte compte) {
    KeyHolder holder = new GeneratedKeyHolder(); //to retreive auto_increment value of pk
    SqlParameterSource parameters = new MapSqlParameterSource()
        .addValue("label", compte.getLabel())
        .addValue("solde", compte.getSolde());
    namedParameterJdbcTemplate.update(INSERT_SQL, parameters, holder);
    compte.setNumero(holder.getKey().longValue()); //store auto_increment pk in instance to return
    return compte;
}

//classe auxiliaire "CompteMapper" pour convertir Resultset jdbc
//en instance de la classe Compte :
class CompteMapper implements RowMapper<Compte> {
    @Override
    public Compte mapRow(ResultSet rs, int rowNum) throws SQLException {
        Compte compte = new Compte();
        compte.setNumero(rs.getLong("numero"));
        compte.setLabel(rs.getString("label"));
        compte.setSolde(rs.getDouble("solde"));
        return compte;
    }
}

```

5. Intégration de JPA/Hibernate dans Spring

6. DAO Spring basé sur JPA (Java Persistence Api)

6.1. Rappel: Entité prise en charge par JPA

```
package entity.persistance.jpa;
//old import (spring5 , JPA2 : javax.persistence.* )
import jakarta.persistence.Column; import jakarta.persistence.Entity;
import jakarta.persistence.Id;      import jakarta.persistence.Table;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
```

```
@Entity
//@Table(name="Compte")
public class Compte {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long numCpt;

    @Column(length=32)
    private String label;

    private double solde;

    public String getLabel() { return this.label; }
    public void setLabel(String label) { this.label=label; }
    //+ autres get/set
}
```

6.2. unité de persistance (persistence.xml facultatif)

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
  http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="myPersistenceUnit"
    transaction-type="RESOURCE_LOCAL">

    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>entity.persistance.jpa.Compte</class>
    <class>entity.persistance.jpa.XxxYyy</class>
    <properties>
        <!-- <property name="hibernate.dialect"
          value="org.hibernate.dialect.MySQL5InnoDBDialect" /> -->
        <property name="hibernate.dialect"
          value="org.hibernate.dialect.H2Dialect" />
        <property name="hibernate.hbm2ddl.auto"
          value="create" /> <!-- or "none" -->
    
```

```
</properties>
</persistence-unit> </persistence>
```

NB : La configuration "Jpa" d'une application spring peut :

- soit être partiellement configurée dans META-INF/persistence.xml et partiellement configurée en mode "spring" (xml ou bien java config)
- soit être entièrement configurée en mode spring (xml , java config) et dans ce cas **le fichier META-INF/persistence.xml peut ne pas exister (il n'est pas absolument nécessaire).**

6.3. Configuration "spring / jpa" classique (en version xml) :

src/mySpringConf.xml

```
<bean id="myDataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost/bibliotheque_db" />
    <property name="username" value="root" /><property name="password" value="root" />
</bean>

<bean id="myEmf"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="myDataSource"/>
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
    </property>
</bean>
```

6.4. TxManager compatible JPA et @PersistenceContext

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="myEmf"/>
</bean>

<tx:annotation-driven transaction-manager="transactionManager" />
```

Cette configuration est indispensable pour que les annotations **@Transactional(readOnly=true)** et **@Transactional(rollbackFor=Exception.class)** qui précèdent les méthodes des services métiers soient prises en compte par Spring de façon à générer (via AOP) une enveloppe transactionnelle.

NB : L'annotation **@PersistenceContext()** d'origine EJB3 permet d'initialiser automatiquement une instance de "entityManager" en fonction de la configuration JPA (META-INF/persistence.xml + entityManagerFactory, ...).

6.5. Configuration Jpa / Spring (sans xml) en mode java-config

La configuration suivante est équivalente aux configurations xml des paragraphes précédents.

```
package tp.myapp.minibank.impl.config;
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource; import java.util.Properties ;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor;
import org.springframework.orm.jpa.vendor.Database;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@EnableTransactionManagement() // "transactionManager" (not "txManager") is expected !!!
@ComponentScan(basePackages={"tp.myapp.minibank","org.mycontrib.generic"})
// for interpretation of @Component , @Controller , ... for @Autowired, @Inject ,...
public class JpaConfig {

    // JpaVendorAdapter (Hibernate ou OpenJPA ou ...)
    @Bean
    public JpaVendorAdapter jpaVendorAdapter() {
        HibernateJpaVendorAdapter hibernateJpaVendorAdapter
            = new HibernateJpaVendorAdapter();
        hibernateJpaVendorAdapter.setShowSql(false);
        hibernateJpaVendorAdapter.setGenerateDdl(false);
        hibernateJpaVendorAdapter.setDatabase(Database.MYSQL);
        // hibernateJpaVendorAdapter.setDatabase(Database.H2);
        return hibernateJpaVendorAdapter;
    }

    // EntityManagerFactory
    @Bean(name="entityManagerFactory")
    public EntityManagerFactory entityManagerFactory(
        JpaVendorAdapter jpaVendorAdapter, DataSource dataSource) {
        LocalContainerEntityManagerFactoryBean factory
            = new LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(jpaVendorAdapter);
        factory.setPackagesToScan("tp.myapp.minibank.persistence.entity");
        factory.setDataSource(dataSource);
        Properties jpaProperties = new Properties(); // java.util
        jpaProperties.setProperty("javax.persistence.schema-generation.database.action",
            "drop-and-create"); // à partir de JPA 2.1
        factory.setJpaProperties(jpaProperties);
    }
}
```

```

factory.afterPropertiesSet();
return factory.getObject();
}

// Transaction Manager for JPA or ...
@Bean(name="transactionManager") //("transactionManager" but not "txManager")
public PlatformTransactionManager transactionManager(
    EntityManagerFactory entityManagerFactory) {
    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory);
    return txManager;
}
}

```

NB : La configuration ci dessus n'a pas besoin de META-INF/persistence.xml

6.6. Simplification "Spring-boot" et @EnableAutoConfiguration

Toute la **configuration** (xml ou bien "java config explicite") des paragraphes précédents peut éventuellement être **considérée comme "pré définie"** lorsque l'on utilise "**spring-boot**" en mode **@EnableAutoConfiguration**.

Les seuls petits paramétrages nécessaires (url , packages à scanner, ...) peuvent être placés dans le fichier **application.properties**

Exemple pour H2

```

# JDBC settings for (h2) embedded DataBase
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:./h2-data/backendApiDb
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.platform=h2
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=create

```

Exemple pour MySQL ou MariaDB

```

spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/backendApiDb
    ?createDatabaseIfNotExist=true&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.hibernate.ddl-auto=none
#spring.jpa.hibernate.ddl-auto=create

```

Exemple pour PostgreSQL

```
#NB: avec postgresql , la base doit exister (même vide) avec username/password  
spring.datasource.driverClassName=org.postgresql.Driver  
spring.datasource.url=jdbc:postgresql://localhost:5432/backendApiDb  
spring.datasource.username=postgres  
spring.datasource.password=root  
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect  
spring.jpa.hibernate.ddl-auto=none  
#spring.jpa.hibernate.ddl-auto=create
```

D'autre part, l'extension facultative (mais très intéressante) "**spring-data**" permet de simplifier énormément le code des "DAO : Data Access Object" (--> voir chapitre "spring-data") .

6.7. DAO «JPA» style «pure JPA,Ejb3» pris en charge par Spring

```

package dao.jpa;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query; ...

@Transactional
@Component //ou @Repository
public class CompteDaoJpa implements CompteDao {

    @PersistenceContext()
    private EntityManager entityManager;

    public List<Compte> getAllComptes() {
        return entityManager.createQuery(
            "Select c from Compte as c", Compte.class)
            .getResultList();
    }

    public Compte getCompteByNum(long num_cpt) {
        return entityManager.find(Compte.class, num_cpt);
    }

    public void updateCompte(Compte cpt) {
        entityManager.merge(cpt);
    }

    public Long createCompte(Compte cpt) {
        entityManager.persist(cpt);
        return cpt.getNumCpt(); //return auto_incr pk
    }

    public void deleteCompte(long numCpt) {
        Compte cpt = entityManager.find(Compte.class, numCpt) ;
        entityManager.remove(cpt);
    }
}

```

VI - Spring-Data (l'essentiel)

1. Spring-Data

L'extension "**Spring-Data**" permet (entre autre) de :

- générer automatiquement des composants "DAO / Repository" modernes (utilisables avec des technologies SQL , NO-SQL ou orientées graphes telles que JPA , MongoDB , Cassandra, Neo4J, ...)
 - accélérer le temps de développement (l'interface suffit souvent, la classe d'implémentation sera générée dynamiquement par introspection et selon certaines conventions).
 - standardiser le format des composants "DAO/Repository" : mêmes méthodes fondamentales.
- On parle alors en termes de "composants DAO consistants" → des automatismes sont possibles (tests en partie automatique ,) .

1.1. Spring-data-commons

"**Spring-data-commons**" est la partie centrale de Spring-data sur laquelle pourra se greffer certaines extensions (pour jpa , pour mongo , ...).

"Spring-data-commons" est essentiellement constituée de **3 interfaces** : **Repository** , **CrudRepository** et **PagingAndSortingRepository** .

- **Repository<T, ID>** n'est qu'une interface de marquage dont toutes les autres héritent.
- **CrudRepository<T, ID>** standardise les méthodes fondamentales (findByPrimaryKey , findAll , save , delete, ...)
- **PagingAndSortingRepository<T, ID>** étend CrudRepository en ajoutant des méthodes supportant le tri et la pagination.

Méthodes fondamentales de **CrudRepository<T ,ID extends Serializable>** :

<code><S extends T> S save(S entity);</code>	Sauvegarde l'entité (au sens saveOrUpdate) et retourne l'entité (éventuellement ajustée/modifiée dans le cas d'une auto-incrémantion ou autre).
<code>Optional<T> findById(ID primaryKey);</code>	Recherche par clef primaire (avec jdk >= 1.8)
<code>Iterable<T> findAll();</code>	Recherche toutes les entités (du type courant/considéré)
<code>Long count();</code>	Retourne le nombre d'entités existantes
<code>void delete(T entity);</code>	Supprime une (ou plusieurs) entités
<code>void deleteById(ID primaryKey);</code>	
<code>void deleteAll();</code>	

<code>boolean exists(ID primaryKey);</code>	Test l'existence d'une entité
---	-------------------------------

NB : principal changement entre "spring-data pour spring 4" et "spring-data pour spring 5" :

Le T `findOne(ID primaryKey);` compatible spring-4 renvoyait auparavant une entité persistante recherchée via sa clé primaire et renvoyait **null** si rien n'était trouvé .

Depuis la version de Spring-data compatible Spring 5 ,

la sémantique de **Optional<T> findOne(T exempleEntity)** consiste à renvoyer une éventuelle entité ayant les mêmes valeurs non-nulles que l'entité exemple passée en paramètre.

Optional<T> findById(ID primaryKey) retourne maintenant une éventuelle entité persistante trouvée (nulle ou pas) dans un objet enveloppe **Optional<T>** qui lui n'est jamais nul .

Le service métier appelant pourra appeler la méthode `.get()` ou bien `.orElse()` de `Optional<T>` de manière à récupérer un accès à l'entité persistante remontée :

```
public Compte rechercherCompte(long num) {
    return daoCompte.findById(num).get(); //retourne exception si null interne
    //return daoCompte.findById(num).orElse(null); //retourne null si null interne
}
```

Variantes de quelques méthodes (surchargées) au sein de CrudRepository :

<code><S extends T> Iterable<S> save(Iterable<S> entities);</code>	Sauvegarde une liste d'entités
<code>Iterable<T> findAll(Iterable<ID> ids);</code>	Recherche toutes les entités (du type considéré) ayant les Ids demandés
<code>void delete(Iterable< ? Extends T> entities)</code>	Supprime une liste d'entités

Rappel : `java.util.Collection<E>` et `java.util.List<E>` héritent de `Iterable<E>`

Fonctionnalité "tri" apportée en plus par l'interface PagingAndSortingRepository :

```
...
Iterable<Personne> personnesTrouvees =
personnePaginationRep.findAll(new Sort(Sort.Direction.DESC, "nom"));
...
```

où `org.springframework.data.domain.Sort` est spécifique à Spring-data .

Fonctionnalité "pagination" apportée en plus par l'interface PagingAndSortingRepository :

```
public void testPagination() {
    assertEquals(10, personnePaginationRep.count());
    Page<Personne> pageDePersonnes =
    // 1re page de résultats et 3 résultats max.
    personnePaginationRep.findAll(new PageRequest(1, 3));
    assertEquals(1, pageDePersonnes.getNumber());
```

```

assertEquals(3, pageDePersonnes.getSize());// la taille d'une page
assertEquals(10, pageDePersonnes.getTotalElements());
assertEquals(4, pageDePersonnes.getTotalPages());
assertTrue(pageDePersonnes .hasContent());

...
}

```

Avec comme types précis :

`org.springframework.data.domain.Page<T>`

et `org.springframework.data.domain.PageRequest` implémentant l'interface
`org.springframework.data.domain.Pageable`

1.2. Spring-data-jpa

Dépendance maven directe (sans spring-boot):

```

<dependencies>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-jpa</artifactId>
    </dependency>
</dependencies>

```

Exemple de version : **1.12.4.RELEASE** (pour spring 4) , **2.0.10.RELEASE** (pour spring 5)

Dépendance maven indirecte (avec spring-boot):

```

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

```

Activation en (rare) configuration xml :

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jpa="http://www.springframework.org/schema/data/jpa"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/data/jpa
                           http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

    <jpa:repositories base-package="com.acme.repositories" />

</beans>

```

Activation en java-config explicite :

```

import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories
...
class Config {}

```

Activation via application.properties (autoConfiguration) :

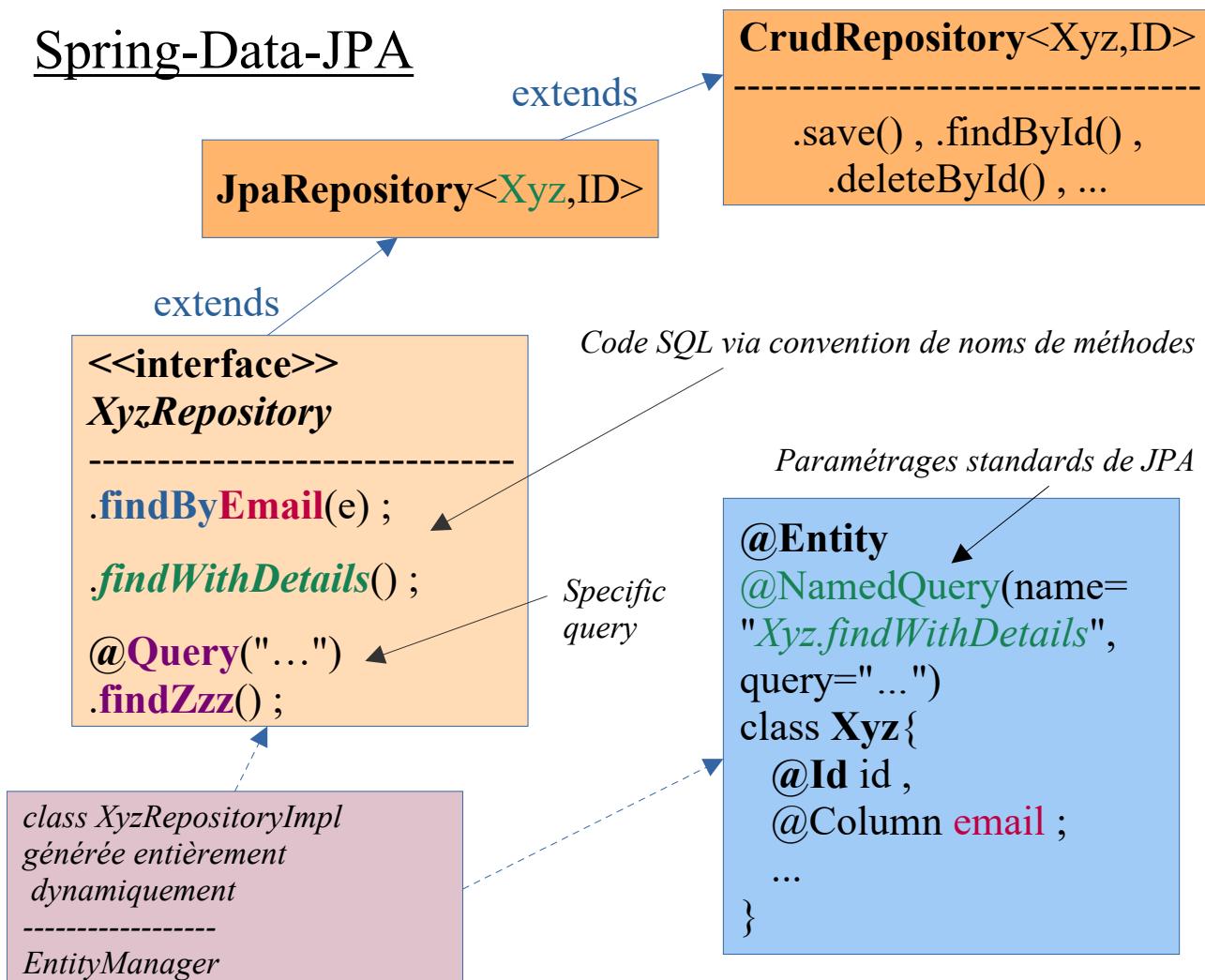
```
spring.data.jpa.repositories.enabled=true
```

Exemple d'interface de DAO/Jpa avec JpaRepository (héritant lui même de CrudRepository) :

```
interface UserRepository extends CrudRepository<User, Long> {
    List<User> findByLastname(String lastname);
}
```

La classe d'implémentation sera générée automatiquement (si `@EnableJpaRepositories` ou si `<jpa:repositories base-package="..." />`)

il suffit d'une injection via `@Autowired` ou `@Inject` pour accéder au composant DAO généré .



Conventions de noms sur les méthodes de l'interface :

find...By, **read...By**, **query...By**, **get...By** and **count...By**,

Exemples :

```
List<User> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
```

// Enables the distinct flag for the query

```
List<User> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
```

```
List<User> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);
```

// Enabling ignoring case for an individual property

```
List<User> findByLastnameIgnoreCase(String lastname);
```

// Enabling ignoring case for all suitable properties

```
List<User> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);
```

// Enabling static ORDER BY for a query

```
List<User> findByLastnameOrderByFirstnameAsc(String lastname);
```

```
List<User> findByLastnameOrderByFirstnameDesc(String lastname);
```

methodNameWithKeyWords(?1,\$2,...)

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname FindByFirstname,	... where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstnameIs , findByFirstnameEquals	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	... where x.age <= ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1

Keyword	Sample	JPQL snippet
GreaterThanOrEqualTo	findByAgeGreaterThanOrEqualTo	... where x.age >= ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull	findByAgeIsNull	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ? 1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	... where x.age not in ?1
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	... where UPPER(x.firstname) = UPPER(?1)

Paramétrage par défaut de JpaRepositories :

CREATE_IF_NOT_FOUND (default) combines CREATE and USE_DECLARED_QUERY

→ on peut donc éventuellement personnaliser l'implémentation des méthodes.

Utilisation de **@NamedQuery** à coté de **@Entity** (ou **<named-query ...>** dans orm.xml) :

Dans orm.xml (référencé par META-INF/persistence.xml ou ...) :

```
<named-query name="User . findByLastname">
    <query>select u from User u where u.lastname = ?1</query>
</named-query>
```

et/ou dans la classe d'entité persistante :

```
@Entity
@NamedQuery(name = "User . findByEmailAddress",
    query = "select u from User u where u.emailAddress = ?1")
public class User {
// ...
}
```

```
public interface UserRepository extends JpaRepository<User, Long>
{
    List<User> findByLastname(String lastname);
    User findByEmailAddress(String emailAddress);
}
```

Utilisation (alternative) de **@Query** (de Spring Data) dans l'interface :

Sémantiquement peut être assez radical pour une interface mais simple et efficace !!!

Exemple :

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query("select u from User u where u.emailAddress = ?1")
    User findByEmailAddress(String emailAddress);
}
```

Variante avec paramètre nommé :

```
@Query("select u from User u where u.emailAddress = :email ")
User findUserByEmailAddress(@Param("email") String emailAddress);
```

@Modifying pour requête modifiant le contenu de la base de données :

```
@Modifying @Query("delete User u where u.active = false")
void deleteDeactivatedUsers();
```

==> et encore beaucoup d'autres possibilités / options dans la **doc de référence de spring-data** .

1.3. Autres parties existantes de Spring-data

Spring-data-Cassandra	
Spring-data-MongoDB	MongoDB en accès synchrone
Spring-data-MongoDB en version réactive	
Spring-data-R2DBC (réactive, asynchrone) R2DBC=Reactive Relational DB Connectivity	Accès aux bases de données relationnelles en mode réactif/asynchrone (pour couplage avec WebFlux)
LDAP , Neo4j , Redis , ElasticSearch	
...	

1.4. Essentiel sur Spring-data-mongo

```
...
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;

@Document("xyz")
public class Xyz {

    @Id
    private String id;
    private String category;
    private String name;

    // @Field(name="price")
    private Double price;
    ...
}
```

```
public interface XyzRepository extends MongoRepository<Xyz, String> {

    // @Query("{name:'?0'}")
    Xyz findByName(String name);

    @Query("{ 'price': { $gte : ?0 , $lte : ?1 } }")
    public List<Xyz> findByPriceBetween(double pMin, double pMax);
}
```

et si nécessaire **@EnableMongoRepositories** sur classe de configuration

application.properties

```
...
#spring.data.mongodb.uri=mongodb+srv://<username>:<pwd>@<cluster>.mongodb.net/xyz
#spring.data.mongodb.database=xyz
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=test
#spring.data.mongodb.username=admin
#spring.data.mongodb.password=password
```

VII - Transactions Spring

1. Support des transactions au niveau de Spring

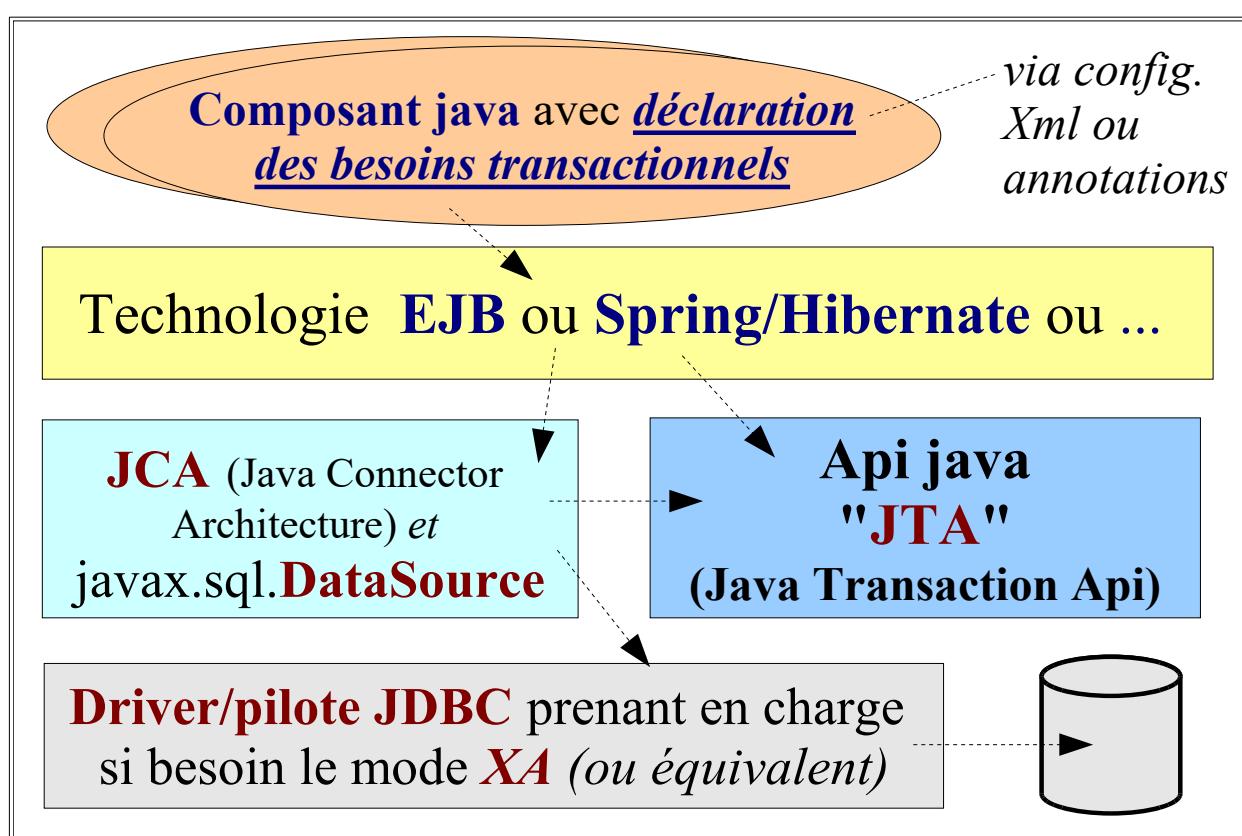
Le framework Spring est capable de gérer (superviser) lui même les transactions devant être menées à bien à partir de certains services applicatifs.

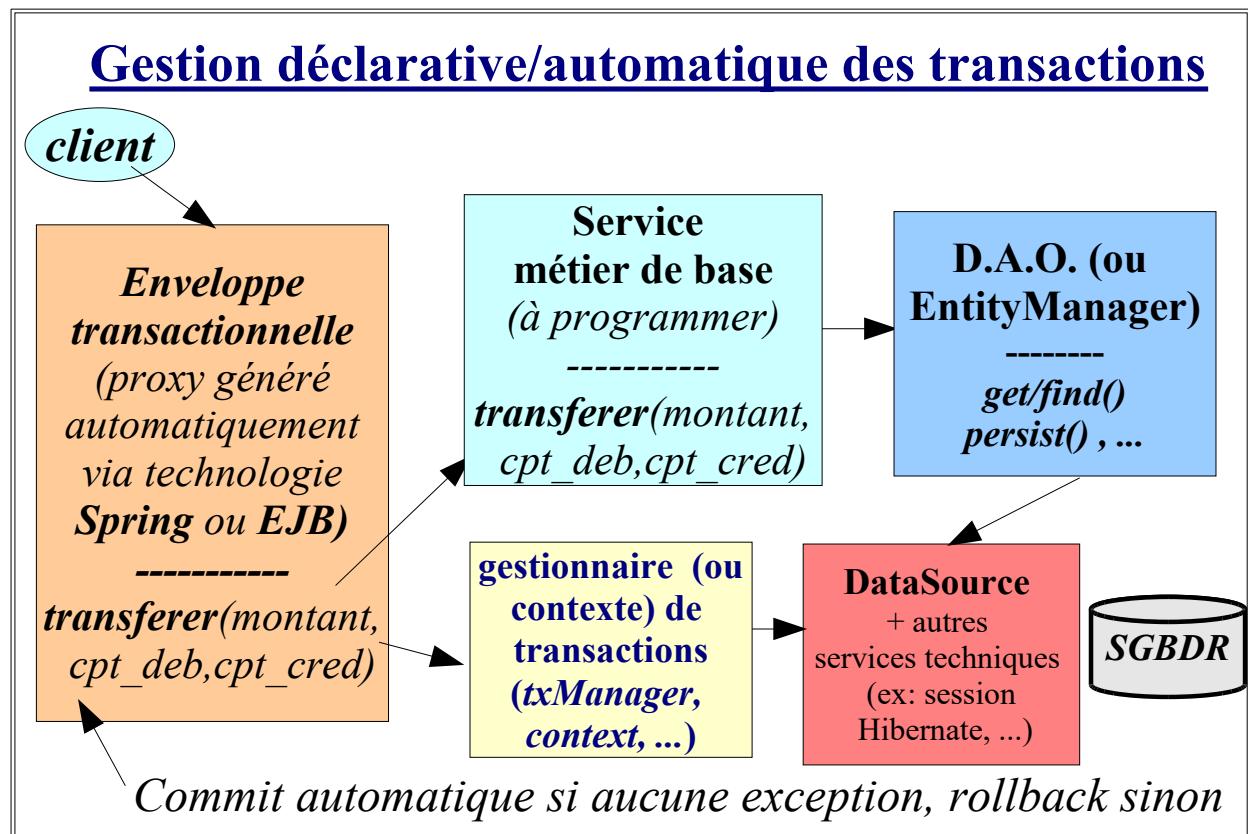
Ceci suppose :

- un paramétrage simple des besoins transactionnels (via xml ou annotations)
- une propagation des ordres transactionnels vers les couches basses (services techniques JDBC , XA , JTA).
-

Etant donné la grande étendue des configurations possibles (JTA ?, Hibernate ?, serveur J2EE ?, EJB ?, ...) les mécanismes transactionnels de Spring doivent être relativement flexibles de façon à pouvoir s'adapter à des situations très variables.

Le composant technique "**txManager**" servira à relayer les ordres de «commit» ou «rollback» vers la source de données (SGBDR) .





L'enveloppe transactionnelle supervisera automatiquement les "commit" et les "rollback" en fonction d'un paramétrage XML (ou bien en fonction de certaines annotations).

Le code généré dans l'enveloppe transactionnelle est à peu près de cette teneur:

```

public void transferer(double montant, long num_cpt_deb, long num_cpt_cred){
    // initialisation (si nécessaire) de la session Hibernate ou de l'entityManager de JPA
    // selon existence dans le thread courant
    tx = ...beginTransaction(); // sauf si transaction (englobante) déjà en cours
    try{
        serviceDeBase.transferer(montant,num_cpt_deb,num_cpt_cred);
        tx.commit(); // ou ... si transaction (englobante) déjà en cours
    }
    catch(RuntimeException ex){    tx.rollback(); /* ou setRollbackOnly(); */ ... }
    catch(Exception e){    e.printStackTrace(); }
    finally{    // fermer si nécessaire session Hibenate ou EntityManager JPA
        // (si ouvert en début de cette méthode)
    }
}

```

2. Propagation du contexte transactionnel et effets

Propagation du contexte transactionnel



Propagation	Tx en cours (appelant)	Tx dans sous service
Required (par défaut)	none tx1	new_tx tx1
Support	none tx1	none tx1
Nested	tx1	sub_tx (in tx1)
...		

NB: Le choix de la propagation peut se faire via `@Transactional(propagation=....)`

Effets de @Transactional (de Spring)

avec
propagation
=Required
(par défaut)



Comportement (engendré par `@Transactional`)

Au début:

Si aucun "entityManager/..." était ouvert au début j'ai dû en ouvrir un.
Si aucune transaction existait auparavant j'ai alors dû en créer une nouvelle .

A la fin:

Je ferme ou finalise que ce que j'ai moi même ouvert/initialisé (tx et/ou ...)
ou bien sinon: simple `tx.setRollbackOnly()` en cas d'exception locale.

3. Configuration du gestionnaire de transactions

3.1. Différentes implémentations de *PlatformTransactionManager*

Principale technologie utilisée au niveau d'un DAO	implémentation de l'interface <i>PlatformTransactionManager</i>
JDBC (jdbcTemplate)	<code>org.springframework.jdbc.datasource.DataSourceTransactionManager</code> avec injection dataSource
JTA	<code>org.springframework.transaction.jta.JtaTransactionManager</code> avec injection dataSource
Hibernate (<i>en direct</i>)	<code>org.springframework.orm.hibernate3.HibernateTransactionManager</code> ou hibernate4/5 avec injection sessionFactory
JPA (over-hibernate)	<code>org.springframework.orm.jpa.JpaTransactionManager</code> avec injection entityManagerFactory

3.2. Exemple de configuration explicite en mode "java-config"

```

@Configuration
@EnableTransactionManagement()
@ComponentScan(basePackages={"tp.myapp.minibank"})
public class ServiceConfig ou JpaConfig{

    ...

    // Transaction Manager for JPA or ...
    @Bean(name="transactionManager")
    public PlatformTransactionManager transactionManager(
        EntityManagerFactory entityManagerFactory) {
        JpaTransactionManager txManager =
            new JpaTransactionManager();
        txManager.setEntityManagerFactory(entityManagerFactory);
        return txManager;
    }

}

```

4. Marquer besoin en transaction avec `@Transactional`

```
<tx:annotation-driven transaction-manager="txManager"/>
```

en config XML

ou bien

```
@EnableTransactionManagement()
```

en mode java-config est nécessaire pour bien interpréter `@Transactional` dans le code d'implémentation des services et des "DAO".

--> Exemple :

```
import org.springframework.transaction.annotation.Transactional;

...
//éventuel @Transactional de niveau classe entière
public class GestionComptesImpl implements GestionComptes {
    ...

    @Transactional(readOnly=true)
    public Compte getCompteByNum(long numCpt) throws MyApplicationException {
        ...
    }

    @Transactional
    public void transferer(long numCompteADebiter, long numCompteACrediter,
                           double montant) throws MyApplicationException {
        ...
    }
}
```

Important:

L'enveloppe transactionnelle générée automatiquement par Spring_AOP ne déclenche par défaut des **rollbacks** que suite à des «unchecked exceptions» (exceptions héritant de `RuntimeException`).

Si l'on souhaite que Spring déclenche des rollback suite à d'autres types d'exceptions, il faut le préciser via le paramètre optionnel **rollbackFor** de l'annotation `@Transactional` (ou de la balise xml `<tx:method>`).

syntaxe générale: `rollbackFor="Exception1,Exception2,Exception3"`.

Exemple: `@Transactional(rollbackFor=Exception.class)`

On peut également choisir le mode de **propagation** du contexte transactionnel via l'attribut propagation de l'annotation `@Transactional` (sachant que la valeur par défaut "**Required**" convient parfaitement dans la majorité des cas).

VIII - Architectures web / spring (vue d'ensemble)

1. Spring web overview

Plein de variantes sont envisageables et on peut quelquefois les faire coexister .

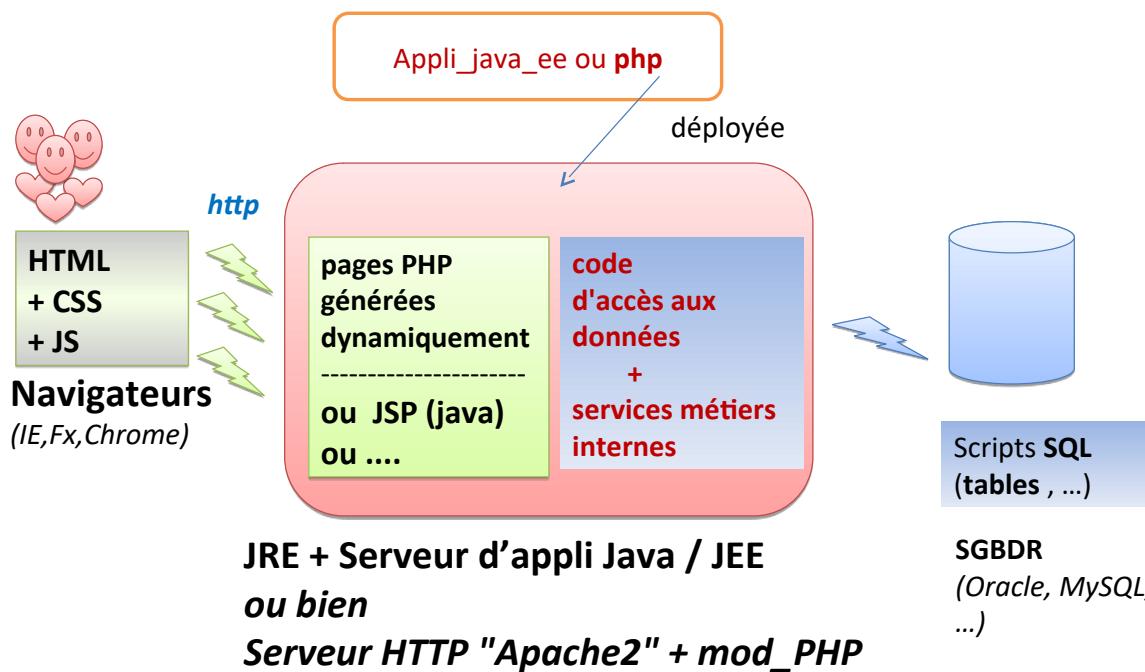
<i>Types de parties web</i>	<i>Caractéristiques importantes</i>
Partie web ne comportant que des WEB-SERVICES REST	@RestController et sécurisation avec des tokens (souvent JWT) quelquefois issus d'un serveur OAuth2/Oidc . Avec souvent une documentation swagger/openApidoc .
Partie web ne comportant que des pages HTML générées dynamiquement (via ".jsp" ou thymeleaf ou autres)	À sécuriser avec des cookies et la protection CRSF . Généralement Codé via : - "Spring + JSF" ou bien - "Spring + Struts 2" ou bien - "Spring-MVC" (avec jsp ou bien thymeleaf) ou bien (rarement) - simples Servlet(s) plus pages JSP
Double partie web (API REST + pages HTML générées dynamiquement)	Prévoir plusieurs points d'entrée et deux types de sécurisation (à faire coexister) !

Dans presque tous les cas il y aura une page d'accueil simple (faisant office de menu) matérialisée sous forme de fichier **index.html** (souvent placée dans **src/main/resources/static**) .

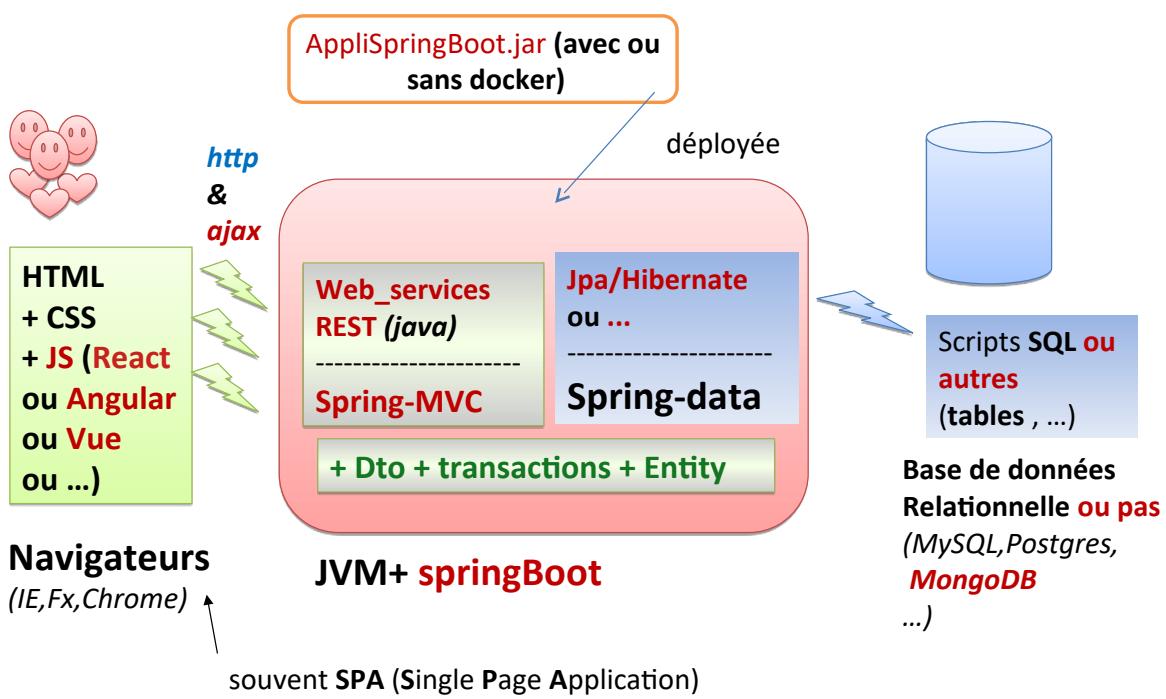
NB :

- Si l'on utilise un framework web additionnel "pas spring" tel que "Struts2" ou "JSF" alors le lien s'effectue via "spring-web" et d'autres compléments facultatifs.
- Le framework "Spring-MVC" peut aussi bien servir à coder une "api REST" qu'à générer dynamiquement des pages HTML (via ".jsp" ou thymeleaf")
- L'application complète spring (avec sa ou ses parties "web") pourra soit fonctionner dans un serveur d'application JEE (ex : tomcat) après un déploiement de ".war" ou bien fonctionner de manière autonome si elle est basée sur springBoot avec un déploiement de ".jar" (placé ou pas dans un conteneur "docker") .
- Depuis "spring >=5" , il est quelquefois possible de coder la partie web en s'appuyant sur des technologies très asynchrones (ceci permet d'un coté d'obtenir un petit gain en performance mais d'un autre coté ça rend le code beaucoup plus complexe à écrire et maintenir d'autant plus que l'on manque encore de recul sur la pérennité des technos asynchrones pas encore standardisées à l'échelle du langage java)

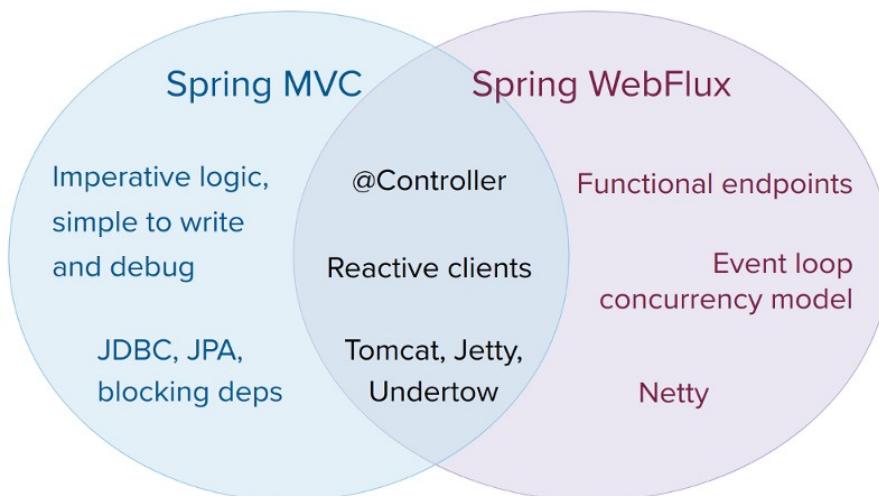
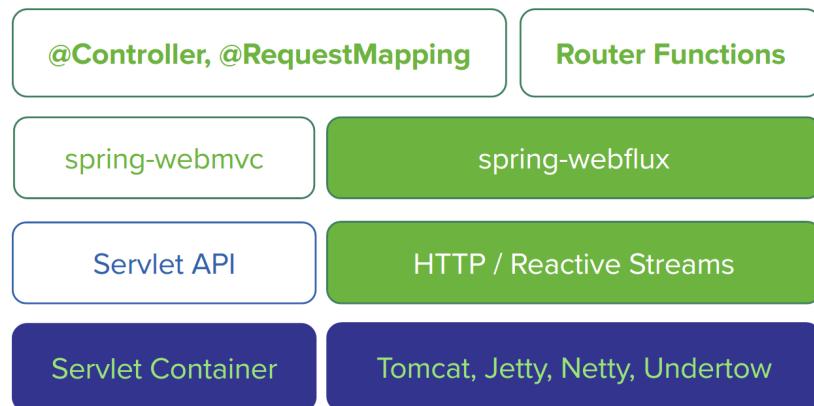
Ancienne architecture web



Frontend + Backend

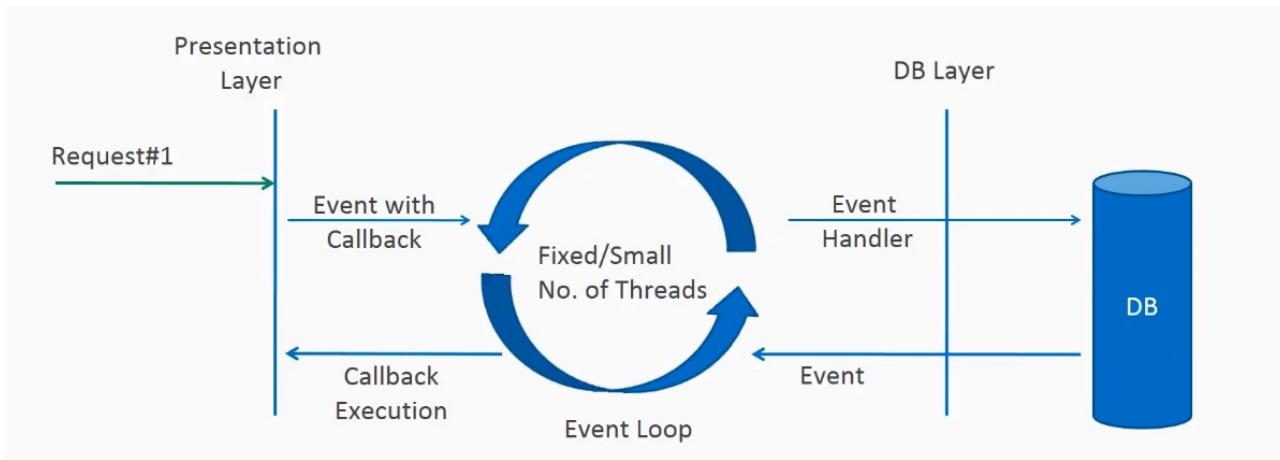


2. Vue d'ensemble sur Spring asynchrone



Framework	Fonctionnalités	Documentation et exemples
Netty	Framework asynchrone non bloquant , peut être utilisé en mode serveur (à la place de tomcat)	https://www.baeldung.com/netty
Reactor	Framework de programmation réactive en java (un peu comme RxJava et RxJs)	https://projectreactor.io/ https://spring.io/reactive https://www.baeldung.com/reactor-core
WebFlux	Une sorte de version asynchrone de Spring-mvc (pouvant être utilisé pour coder ou invoquer des api REST)	https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html https://www.baeldung.com/spring-webflux

Attention: *Nouveautés de version 5 (technologies très récentes, pas encore "classique/mature").*



En gros, Spring web-flux reprend les mêmes principes de fonctionnement que nodeJs .

IX - Spring-web (avec serveur, génération HTML)

1. Lien avec un serveur JEE et Spring web

Configuration nécessaire pour qu'une application Spring (packagée comme un ".war") puisse bien démarrer au sein d'un serveur Jakarta-EE (ex : tomcat10) :

Type d'application Spring	Point d'entrée en mode "web" au sein d'un serveur JEE (après déploiement du ".war")
Spring-framework (sans SpringBoot) et avec veille config mySpringConf.xml et WEB-INF/web.xmml	<pre><context-param> <param-name>contextConfigLocation</param-name> <param-value>classpath:/mySpringConf.xml</param-value> </context-param> <listener> <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class> </listener></pre>
Spring-framework (sans SpringBoot) et avec config java	<p>Classe java implémentant l'interface WebApplicationInitializer et comportant une méthode onStartup() au sein de laquelle on enregistre la configuration Spring dans un ContextLoaderListener associé au servletContext .</p> <p>Ou bien variante simplifiée en utilisant la classe abstraite AbstractContextLoaderInitializer .</p>
SpringBoot moderne	Faire hériter la classe principale de SpringBootServletInitializer et coder .configure() avec builder. source(...)

2. Injection de Spring au sein d'un framework WEB

2.1. WebApplicationContext (configuration xml)

Cette ancienne configuration (au format XML) est placée dans un document complémentaire à cette version récente du support de cours

2.2. WebApplicationContext (configuration java-config)

```
class MyWebApplicationInitializer implements WebApplicationInitializer {
    public void onStartup(.. servletContext )...{
        WebApplicationContext context = new AnnotationConfigWebApplicationContext();
        context.register (MyWebRootAppConfig.class );
        servletContext.addListener (new ContextLoaderListener (context ));
        //...
    }
}
```

}

MyWebRootAppConfig.class peut par exemple correspondre à la classe de configuration Spring principale (elle même potentiellement reliée à d'autres sous-configurations via @Import).

Variante simplifiée via AbstractContextLoaderInitializer

```
public class AnnotationsBasedApplicationInitializer  
    extends AbstractContextLoaderInitializer {  
  
    @Override  
    protected WebApplicationContext createRootApplicationContext() {  
        AnnotationConfigWebApplicationContext rootContext  
            = new AnnotationConfigWebApplicationContext();  
        rootContext.register(MyWebRootAppConfig.class);  
        return rootContext;  
    }  
}
```

URL pour approfondir si bseoin le sujet :
<https://www.baeldung.com/spring-web-contexts>

2.3. WebApplicationContext (accès et utilisation)

Au sein d'un servlet ou bien d'un élément annexe on peut instancier des Beans via Spring :

```
application = .... getServletContext(); // application prédéfini au sein d'une page JSP  
WebApplicationContext ctx =  
    WebApplicationContextUtils.getWebApplicationContext(application);  
IXxx bean = (IXxx) ctx.getBean(...);  
....  
request.setAttribute("nomBean",bean); // on stocke le bean au sein d'un scope (session,request,...)  
rd.forward(request,response); // redirection vers page JSP
```

NB : Spring-web propose en plus des configurations complémentaires spécifiques pour bien intégrer la plupart des frameworks java-web (Struts, JSF , ...)

3. Packaging d'une application SpringBoot pour un déploiement vers un serveur JEE

Par défaut, une application moderne SpringBoot est packagée en tant que ".jar" et fonctionne de manière autonome sans serveur (en embarquant un conteneur web imbriqué tel que tomcat ou jetty).

Il est tout de même possible de configurer une application SpringBoot de manière à ce que l'on puisse effectuer un déploiement web classique dans un serveur d'application JEE.

Il faut pour cela :

- modifier le **packaging** de l'application dans pom.xml (de "jar" vers "**war**")

`<packaging>war</packaging>`
- faire en sorte que la classe principale de l'application hérite de **SpringBootServletInitializer**
- déclencher **builder.sources(MySpringBootApplication .class)**; dans une redéfinition de **configure()** de manière à préciser le point d'entrée de la configuration de l'application.

Exemple :

```
package tp.appliSpring;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.support.SpringBootServletInitializer;

@SpringBootApplication
public class MySpringBootApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return builder.sources(MySpringBootApplication.class);
        /* .profiles("dev,reInit"); */ //setting profiles here
        // or with system properties of the server (ex: tomcat)
    }

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

Si le .war construit par maven est *xyz.war* , et que ce .war est recopié dans le sous répertoire "**webapps**" de tomcat , l'URL par défaut au sein de tomcat est alors <http://localhost:8080/xyz>

NB : si besoin *set JAVA_HOME=path_to_jdk17plus* au sein de tomcat.../bin/startup.bat

4. Présentation du framework "Spring MVC"

"Spring Web MVC" est une partie optionnelle du framework spring servant à gérer la logique du design pattern "MVC" dans le cadre d'une intégration "spring".

A l'origine (vers les années 2005-2012), "Spring MVC" était à voir comme un petit framework java/web (pour le côté serveur) qui se posait comme une alternative à Struts2 ou JSF2.

Plus récemment, "Spring MVC" (souvent intégré dans SpringBoot) est énormément utilisé pour développer des Web-Services REST et est quelquefois encore un peu utilisé pour générer des pages HTML (via des vues ".jsp" ou bien des vues ".html" de Thymeleaf).

4.1. configuration maven pour spring-mvc

Dépendances maven nécessaires (en intégration moderne "spring-boot"):

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

et (si vues de type ".jsp")

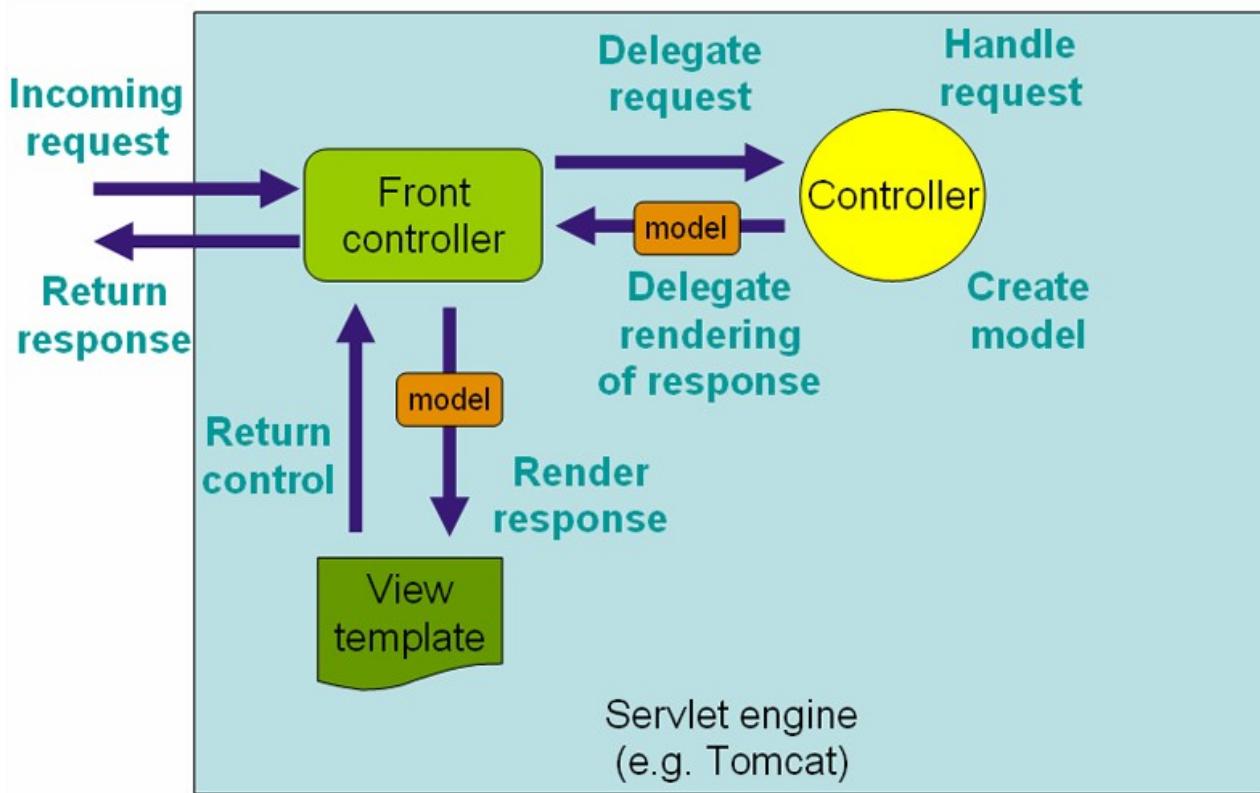
```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>jakarta.servlet.jsp.jstl</groupId>
    <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
</dependency>
<dependency>
    <groupId>org.glassfish.web</groupId>
    <artifactId>jakarta.servlet.jsp.jstl</artifactId>
</dependency>
<!-- ou ancien équivalent spring5/springBoot2/jee -->
<!-- <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
    </dependency> -->
```

ou bien (avec Thymeleaf)

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>nz.net.ultraq.thymeleaf</groupId>
    <artifactId>thymeleaf-layout-dialect</artifactId>
</dependency>
```

5. Mécanismes fondamentaux de "spring mvc"

5.1. Principe de fonctionnement de SpringMvc :



NB :

- Le **contrôleur** est une instance d'une classe java préfixée par `@Controller` (composant spring de type contrôleur web) et de `@Scope("singleton")` par défaut .
Ce contrôleur a la responsabilité de préparer des données (souvent récupérées en base et quelquefois à partir de critères de recherches)
- Le **model** est une table d'association (nomAttribut, valeurAttribut) (par défaut en `scope=request`) permettant de passer des objets de valeurs à afficher au niveau de la vue.
- La **vue** est responsable d'effectuer un rendu (souvent "html + css + js") à partir des valeurs du modèle.
La **vue** est souvent une **page JSP** ou bien un **template "Thymeleaf"** .
- Ce framework peut fonctionner en mode simplifié (sans vue) avec une génération automatique de réponse au format **JSON** ou **XML** dans le cadre de Web Services REST.

5.2. Exemple Spring-Mvc simple en version ".jsp":

src/main/resources/application.properties

```
server.servlet.context-path=/myMvcSpringBootApp
server.port=8080
#spring.mvc.view.prefix=/WEB-INF/view/
spring.mvc.view.prefix=/jsp/
spring.mvc.view.suffix=.jsp
```

Avec cette configuration , un **return "xy"** d'un contrôleur déclenchera l'affichage de la page **/jsp/xy.jsp** et selon la structure du projet , le répertoire **/jsp** sera placé dans **src/main/resources/META-INF/resources** ou ailleurs .
(NB : si projet sans springBoot , dans **src/main/webapp**).

Exemple élémentaire :

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public String helloWorld(Model model) {
        model.addAttribute("message", "Hello World!");
        return "showMessage";
    }
}
```

Au niveau de **/jsp/showMessage.jsp**, l'affichage de message pourra être effectué via **\${message}**.

```
<html>
<head><title>showMessage</title></head>
<body>
    <p>message=<b>${message}</b></p>
</body>
</html>
```

X - Api REST via spring-Mvc et OpenApiDoc

1. Web services "REST" pour application Spring

Pour développer des Web Services "REST" au sein d'une application Spring , il y a deux possibilités distinctes (à choisir) :

- s'appuyer sur l'API standard **JAX-RS** et choisir une de ses implémentations (**CXF3** ou **Jersey** ou ...)
- s'appuyer sur le framework "**Spring web mvc**" et utiliser **@RestController** .

La version "JAX-RS standard" nécessite pas mal de librairies (jax-rs, jersey ou cxf , jackson et tout un tas de dépendances indirectes) .

La version spécifique spring nécessite un peu moins de librairies (spring-web , spring-mvc , jackson) et s'intègre mieux dans un écosystème spring (spring-security ,).

Exemples de dépendances "maven" sans spring-boot :

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.3.RELEASE</version>
    <scope>compile</scope>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.10.2</version> <!-- to produces json -->
</dependency>
...
```

Dépendances "maven" indirecte (avec spring-boot) :

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

Dans *application.properties* :

```
server.servlet.context-path=/webappXy ou ...
server.port=8181 ou 8080 ou ...
```

2. Variantes d'architectures

Un service web doit absolument retourner des structures de données stables (DTO = Data Transfert Object) idéalement indépendantes des structures des entités persistantes de manière à limiter les adhérences entre couches logicielles et garantir une bonne évolutivité .

Le code des conversions entre DTO et entités persistantes s'effectue souvent au sein d'un composant utilitaire "Converter" ou "ModelMapper" .

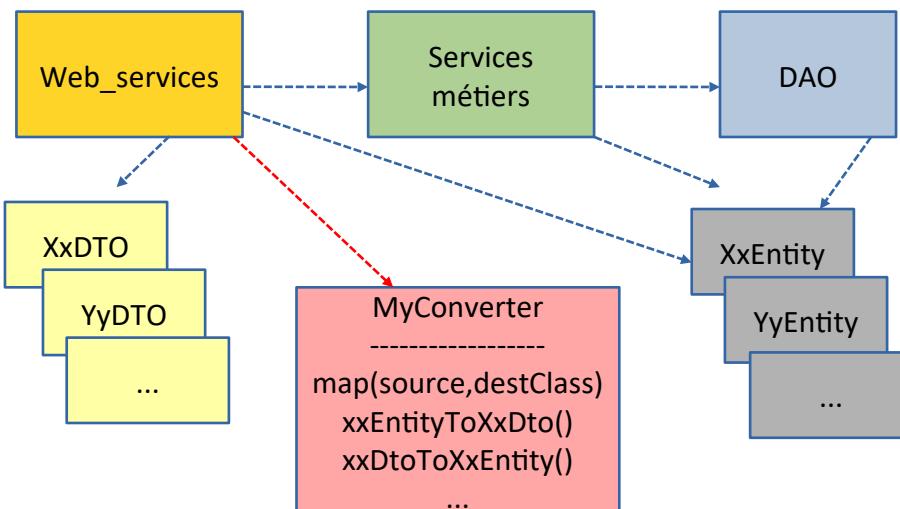
Ce convertisseur peut être codé de des manières suivantes :

- entièrement manuellement (méthodes xxEntityToXxDto() ,)
- en s'appuyant sur la très bonne technologie "**MapStruct**" (très performante) décrite en annexes
- en s'appuyant sur BeanUtils.copyProperties()
- en s'appuyant sur la technologie "modelMapper" (moins performante mais plus générique)
- sur une combinaison "maison" des technologies précédentes

Le déclenchement des conversions peut être opéré/déclenché à différents niveaux :

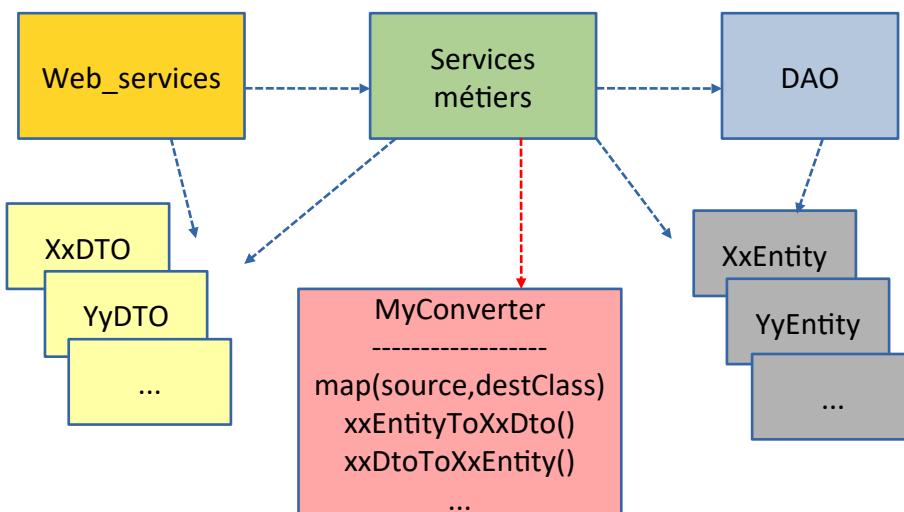
- au niveau des web services
- au niveau des services internes
- au niveau des adaptateurs de persistance (dans le cadre d'une architecture hexagonale)
- éventuellement à plusieurs niveaux

Conversion DTO sur WebService



pour cas "extrêmement simples" seulement.

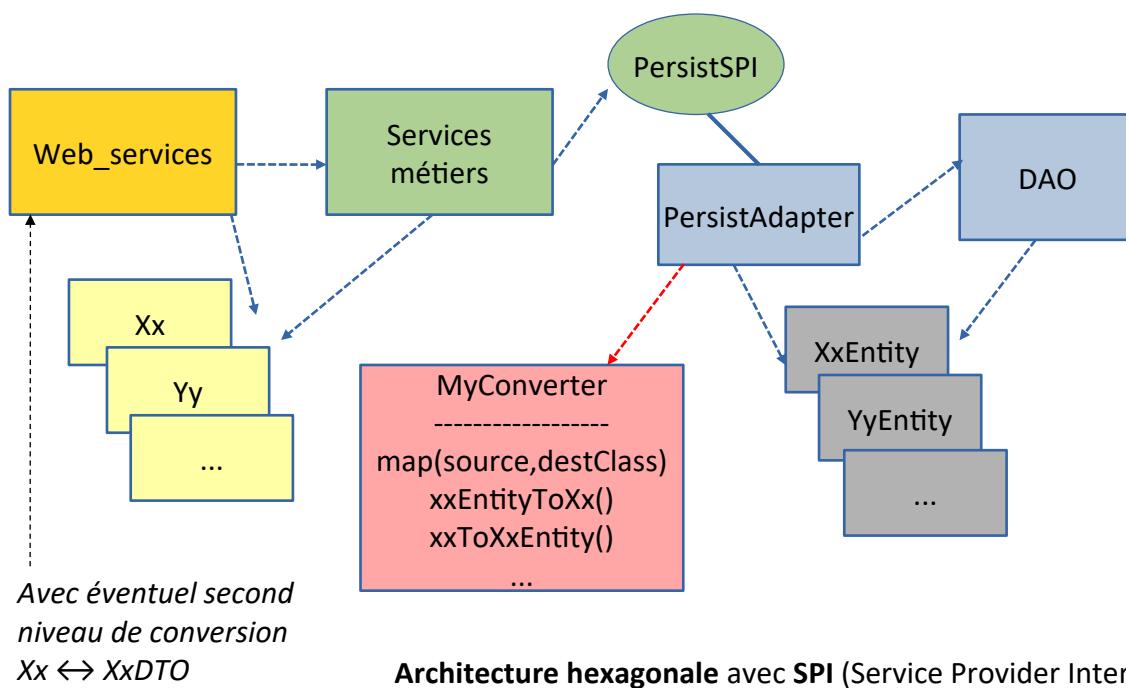
Conversion DTO sur Services métiers



Lorsque la conversion est effectuée en mode "n-tiers" au niveau des services métiers internes, alors :

- l'**interface du service** expose des **DTO** (ou autre) au niveau des paramètres d'entrée et au niveau des valeurs de retour des méthodes publiques
- le code interne du service métier (souvent préfixé par `@Service`) utilise des éléments **privés** de type "**DAO/repository**" et "**Entity**" avec des **conversions** à déclencher .

Conversion Entity sur adaptateur SPI



Le schéma ci-dessus correspond à une vision extrêmement simplifiée de l'architecture hexagonale (complexe et facultative).

Au sein de cette architecture , on a :

- en plein milieu la zone "domaine métier" (avec ses données métiers "Xx" , "Yy" et ses services métiers)

- en périphérie des zones "d'infrastructures" (persistance , échanges "kafka" ,) qui doivent s'adapter aux interfaces entrantes et sortantes (SPI) de la zone centrale "domaine métier" .

Dans un tel cadre , les adaptateurs de persistance peuvent prendre en charge des conversions entre "Xx" et "XxEntity" et la partie "Api REST" peut soit réutiliser les classes "Xx" telles quelles si elles semblent très stables ou bien opérer un second niveau de conversion "Xx" vers "XxDTO" .

NB :

- Tous les schémas précédents sont eux-même sujets à de multiples variantes (design-patterns aux multiples implémentations possibles, ...).
Sources de variantes : "record" ou "lombok" , généricité/héritage , etc ...
- Comme souvent un bon compromis "simplicité/fonctionnalités" doit être déterminé en fonction de la taille et de la complexité de l'application (KISS : Keep It Simple Stupid , ...)

3. WS REST via Spring MVC et @RestController

L'annotation fondamentale **@RestController** (héritant de **@Controller** et de **@Component**) déclare que la classeRestCtrl correspond à l'implémentation "spring-mvc" d'un composant de l'application de type "Contrôleur de Web Service REST".

On a par défaut **@ResponseBody** avec **@RestController** et cela signifie que la valeur de retour d'une des méthodes publiques du contrôleur sera quasi directement renvoyée au client http (sans passer par une page JSP ni un autre type de vue) .

Cependant , Lorsque la valeur de retour sera un *objet java , celui ci sera automatiquement transformé en JSON* (ou autre) avant d'être retourné au client http (ex : code js / appel ajax)

3.1. Gestion des requêtes en lecture (mode GET)

Exemple :

DeviseJsonRestCtrl.java

```
package tp.app.zz.web.rest;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;
...
@RestController
@RequestMapping(value="/rest/devise-api/v1/devises" ,
    headers="Accept=application/json")
public class DeviseJsonRestCtrl {

    @Autowired //ou @Inject
    private ServiceDevises serviceDevises; //internal business service or DAO

    //RECHERCHE UNIQUE selon RESOURCE-ID:
    //URL de déclenchement: .../webappXy/rest/devise-api/v1/devises/EUR
    @GetMapping("/{codeDevise}")
    public Devise getDeviseByCode(@PathVariable("codeDevise") String codeDevise) {
        return serviceDevises.getDeviseByCode(codeDevise);
    }

    //RECHERCHE MULTIPLE :
    //URL de déclenchement: webappXy/rest/devise-api/v1/devises
    //ou webappXy/rest/devise-api/v1/devises?changeMini=1
    @GetMapping()
    public List<Devise> getDevisesByCriteria(
        @RequestParam(value="changeMini",required=false) Double changeMini) {
        if(changeMini==null)
            return serviceDevises.getAllDevises();
        else
            return serviceDevises.getDevisesByChangeMini(changeMini);
    }
}
```

NB :

@RequestParam avec required=false si paramètre facultatif en fin d'URL

Si l'ensemble de la classe java préfixée par @RestController comporte

@RequestMapping(value="....", headers="Accept=application/json")

alors par défaut les valeurs en retour des méthodes publiques préfixées par **@RequestMapping** seront automatiquement converties au format **JSON** (en s'appuyant en interne sur la technologie **jackson-databind**) .

Techniquement possible mais très rare : retour direct d'une simple "String" (**text/plain**) :

```
//URL : .../devises/convert?amount=50&src=EUR&target=USD
@RequestMapping(value="/convert", method=RequestMethod.GET,
    headers="Accept=text/plain")
//@ResponseBody par défaut avec @RestController
String convert(@RequestParam("amount") double amount,
    @RequestParam("src") String src,
    @RequestParam("target") String target) {
    double sommeConvertie=gestionDevises.convertir(amount, src, target);
    System.out.println("sommeConvertie="+sommeConvertie);
    return String.valueOf(sommeConvertie);
}
```

==> L'exemple ci-dessus est très déconseillé sur une api REST .

Un format de retour homogène (XML ou très souvent JSON) est en général attendu à la place .

3.2. @RequestBody et ReponseEntity<T>

NB : il est techniquement possible de convertir explicitement une "Json String" en objet java via l'api "jackson" comme le montre l'exemple inutilement long suivant (à ne pas reproduire , juste pour montrer certains mécanismes internes):

```
...
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMethod;
import com.fasterxml.jackson.databind.DeserializationFeature;
import com.fasterxml.jackson.databind.ObjectMapper;

@RestController
@RequestMapping(value="/rest/devise-api/v1/devises",
    headers="Accept=application/json")
public class DeviseJsonRestCtrl {

    ...
    @PutMapping()
    Devise updateDevise(@RequestBody String deviseAsString, ...) {
        Devise devise=null;
        try {
            ObjectMapper jacksonMapper = new ObjectMapper();
            jacksonMapper.configure(
                DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
            devise = jacksonMapper.readValue(deviseAsString, Devise.class);
            System.out.println("devise to update:" + devise);
            serviceDevises.updateDevise(devise);
        }
    }
}
```

```

        return devise;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
} ....}

```

Ceci dit , Spring-Mvc est capable d'effectuer de lui même automatiquement cette conversion.
L'écriture suivante (plus simple, à reproduire) assure les mêmes fonctionnalités :

```

@RestController
@RequestMapping(value="/rest/devise-api/v1/devises",
                 headers="Accept=application/json")
public class DeviseJsonRestCtrl {
...
@PutMapping()
ResponseEntity<Devise> updateDevise(@RequestBody Devise devise , ...) {
    System.out.println("devise to update:" + devise);
    serviceDevises.updateDevise(devise);
    return ...;
} ....
}

```

NB : dans tous les cas , il sera quelquefois nécessaire de contrôler le comportement des "sérialisations/dé-sérialisations java <--> json" en incorporant certaines annotations de "jackson" au sein des classes de données (dto / payload) à véhiculer.

A ce sujet , l'annotation **@JsonIgnore** (sémantiquement équivalent à **@XmlTransient**) peut quelquefois être utile pour limiter la profondeur des données échangées .

Apport important de la version 4 : ResponseEntity<T>

Depuis "Spring4" , une méthode d'un web-service REST peut éventuellement retourner une réponse de Type **ResponseEntity<T>** ce qui permet de **retourner d'un seul coup**:

- un statut (OK , NOT_FOUND , ...)
- le corps de la réponse : objet (ou liste) de type T convertie en json
- un éventuel "header" (ex: url avec id si auto_incr lors d'un POST)

Exemple:

```

@GetMapping("/{codeDev}")
ResponseEntity<Devise> getDeviseByName(@PathVariable("codeDev") String codeDevise) {
    Devise dev = gestionDevises.getDeviseByPk(codeDevise);
    if(dev!=null)
        return new ResponseEntity<Devise>(dev, HttpStatus.OK);
    else
        return new ResponseEntity<Devise>(HttpStatus.NOT_FOUND);//404
}

```

ou bien

```
ResponseEntity< ?> getDeviseByName(...){  
....  
    else  
        return new ResponseEntity<String> ("{ \"err\" : \"devise not found\"}",  
                                         HttpStatus.NOT_FOUND) ;//404  
}
```

Autre exemple (ici en mode **DELETE**) :

```
//url : ...webappXy/rest/devise-api/v1/devises/EUR  
@DeleteMapping("/{codeDev}")  
public ResponseEntity< ?> deleteDeviseByCode(@PathVariable("codeDev")String codeDevise){  
    deviseService.deleteDeviseByCode(codeDevise);  
    return new ResponseEntity< ?>(HttpStatus.NO_CONTENT);  
    //NO_CONTENT = 204 = OK mais sans message  
    //ou bien return ResponseEntity.ok(  
    //new MessageDto("devise with code=" + codeDevise + " successfully deleted"));  
    //200/OK + message  
    //exception handler may return NOT_FOUND or INTERNAL_SERVER_ERROR  
}
```

NB : Bien que très finement paramétrable , un **return new ResponseEntity<?>** sera généralement moins astucieux qu'un simple **throw new ...ClasseException** gérée par un **ResponseTypeExceptionHandler** plus simple et plus efficace (ce sera vu dans un paragraphe ultérieur)

Eventuelles variations (équivalences):

@GetMapping(...) est équivalent à **@RequestMapping(... , method=RequestMethod.GET)**
@PostMapping(...) est équivalent à **@RequestMapping(... , method=RequestMethod.POST)**
@PutMapping(...) est équivalent à **@RequestMapping(... , method=RequestMethod.PUT)**
@DeleteMapping(...) équivaut à **@RequestMapping(..., method=RequestMethod.DELETE)**

3.3. Variantes de code :

Ne retourner qu'un statut :

```
return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
```

Retourner "statut" plus un message en JSON :

```
return ResponseEntity.ok(new MessageDto("compte with id=" + id + " successfully deleted"));  
return ResponseEntity.ok().headers(...).body(...);
```

Quelques équivalences :

- new ResponseEntity< ?>(HttpStatus.NOT_FOUND)
- ResponseEntity.status(HttpStatus.NOT_FOUND).build()
- ResponseEntity.notFound().build()

avec ou sans Optional :

```
@GetMapping("/{id}")  
public ResponseEntity<?> getXyzById(@PathVariable("id") long id) {  
    Optional<Xyz> xyzOptional = serviceXyz.rechercherXyzOptional(id);  
    /*  
     * return xyzOptional.map( ResponseEntity::ok  
     * .orElse(ResponseEntity.notFound().build());  
     */  
    return ResponseEntity.of(xyzOptional); //écriture équivalente plus compacte  
}
```

avec ou sans record (java 17+):

SpringMvc et jackson-databind savent bien gérer les "**record**" au sein des versions récentes.
Ceux ci sont exploitables sur des **DTOs** .

Exemple de code pour "POST" et "PUT" :

```
//appelé en mode POST
//avec url = http://localhost:8181/appliSpring/rest/api-xyz/v1/xyz
//avec dans la partie "body" de la requête { "id" : null , "label" : "...." , "price" : 50.0 }
@PostMapping("")
public ResponseEntity<?> postXyz(@Valid @RequestBody XyzToCreate obj) {
    Xyz savedObj = serviceXyz.create(obj); //avec id auto_incrémenté

    URI location = ServletUriComponentsBuilder
        .fromCurrentRequest()
        .path("/{id}")
        .buildAndExpand(savedObj.getId()).toUri();
    //return ResponseEntity.created(location).build();
    //return 201/CREATED , no body but URI to find added obj

    return ResponseEntity.created(location).body(savedObj);
    //return 201/CREATED with savedObj AND with URI to find added obj

    /* ou bien encore return ResponseEntity.ok()
       .headers(responseHeadersWithLocation).body(savedObj);
    */
}
}
```

Résultat :

201	{ "id": 5, "label": "....", "price": 50 }	content-type: application/json location: http://localhost:8181/.../xyz/5
-----	---	---

```
//à appeler en mode PUT
//avec url = http://localhost:8181/appliSpring/rest/api-xyz/v1/xyz/1
//avec dans la partie "body" de la requête { "id" : 1 , "label" : "..." , "price" : 120.0 }
@PutMapping("/{id}")
public ResponseEntity<Xyz> putCompte(@RequestBody Xyz obj,
                                      @PathVariable("id") Long idToUpdate) {
    obj.setId(idToUpdate);
    Xyz updatedObj = serviceXyz.update(obj);
    return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
    //204 : OK sans aucun message dans partie body

    //exception handler may return NOT_FOUND or INTERNAL_SERVER_ERROR
}
```

Résultat :

204		
404	{ "status": "NOT_FOUND", "timestamp": "2024-12-11 01:36:37", "message": "entity not found with id=56"}	content-type: application/json

3.4. Réponse et statut http par défaut en cas d'exception

Si une méthode d'un contrôleur REST remonte une exception java qui n'est pas rattrapée par un try/catch , la technologie Spring-Mvc retourne alors (selon configuration et contexte) une réponse et un statut HTTP par défaut de ce type:

```
{
  "timestamp": 152....56,
  "status": 500 ,
  "error": "Internal Server Error",
  "exception": "java.lang.NullPointerException",
  "message": ".....",
  "path": "/rest/devise/67573567" }
```

Le statut HTTP retourné par défaut dans l'entête de la réponse en cas d'exception est généralement **500 (INTERNAL_SERVER_ERROR)** .

3.5. @ResponseStatus

Dans le cadre d'une remontée d'exception personnalisée il est possible de préciser le statut HTTP (pas systématiquement 500) qui sera remonté via l'annotation **@ResponseStatus()**

Exemple :

```
@ResponseStatus(HttpStatus.NOT_FOUND) //404
public class MyEntityNotFoundException extends RuntimeException{
    public MyEntityNotFoundException(String message) {
        super(message);
    }
    ...
}
```

Un appel HTTP avec une URL finissant (avec une erreur ici volontaire) par "/devise/**EURy**"

---> renvoie **404** et un message d'erreur au format JSON/spring-Web-MCV HOMOGENE :

```
{
  "timestamp": "2020-02-03T17:23:45.888+0000",
  "status": 404,
  "error": "Not Found",
  "message": "echec suppression devise pour codeDevise=EURy",
  "trace": "org.mycontrib.backend.exception.MyEntityNotFoundException:.....",
  "path": "/spring-boot-backend/rest/devise-api/private/role_admin/devise/EURy"
}
```

C'est correct . Cependant le mécanisme **ResponseEntityExceptionHandler** (qui sera présenté ci après) sera encore plus perfectionné et donc généralement préférable.

3.6. ResponseEntityExceptionHandler (très bien)

@RestController avec ExceptionHandler

```

@RestController
@RequestMapping(...)
class XyzCtrl{

@GetMapping(...)
Xyz getById(...){
    return xyzService.searchById(id);
}

@PostMapping()
ResponseEntity postXyz(
    @RequestBody xyz){
....}
...
}
  
```

@ControllerAdvice

```

class MyExHandler extends
ResponseEntityExceptionHandler {

    @Override handle...(){...}

    @ExceptionHandler(MyNotFoundException.class)
    handleNotFoundEx(...ex){
        return ResponseEntity.... ;
        //with 404 and message from ex
    }
}
  
```

Sorte de try/catch par défaut appliquée automatiquement en cas de besoin

Appel sans try/catch

@Service
@Transactional

```

class XyzService{
    ▲ searchById() throws
    MyNotFoundException {...}
    ...
}
  
```

```
class MyNotFoundException extends RuntimeException{ ...}
```

ApiError.java (DTO for custom error message)

```

...
//@Getter @Setter @ToString @NoArgsConstructor
public class ApiError {
    private HttpStatus status;
    private String message;
    //private String details;
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd hh:mm:ss")
    private LocalDateTime timestamp;

    public ApiError(HttpStatus status, String message) {
        super();
        this.status = status;    this.message = message;
        this.timestamp = LocalDateTime.now();
    }
...
}
  
```

RestResponseEntityExceptionHandler.java

```

package tp.appliSpring.generic.rest;

import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.http.converter.HttpMessageNotReadableException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;
import tp.appliSpring.core.exception.ConflictException;
import tp.appliSpring.core.exception.NotFoundException;
import tp.appliSpring.dto.ApiError;

@ControllerAdvice
public class RestResponseEntityExceptionHandler
    extends ResponseEntityExceptionHandler {

    private ResponseEntity<Object> buildResponseEntity(ApiError apiError) {
        return new ResponseEntity<>(apiError, apiError.getStatus());
    }

    @Override
    protected ResponseEntity<Object>
    handleHttpMessageNotReadable(HttpMessageNotReadableException ex,
        HttpHeaders headers, HttpStatus status, WebRequest request) {
        String error = "Malformed JSON request";
        return buildResponseEntity(new ApiError(HttpStatus.BAD_REQUEST, error, ex));
    }

    @ExceptionHandler(NotFoundException.class)
    protected ResponseEntity<Object> handleEntityNotFound(
        NotFoundException ex) {
        return buildResponseEntity(new ApiError(HttpStatus.NOT_FOUND,ex));
    }

    @ExceptionHandler(ConflictException.class)
    protected ResponseEntity<Object> handleConflict(
        ConflictException ex) {
        return buildResponseEntity(new ApiError(HttpStatus.CONFLICT,ex));
    }
}

```

Et grâce à cela les exceptions java retournées par les services et contrôleurs REST :

- n'ont plus besoin d'être décorées par `@ResponseStatus` → meilleures séparation des couches
- seront automatiquement transformées en messages très personnalisés et accompagnés du bon statut HTTP .

3.7. Validation des valeurs entrantes (@Valid)

Dans le cadre d'un échec de validation de la requête avec **@Valid** sur le paramètre d'entrée d'une méthode d'un contrôleur REST et avec des annotations de javax.validation (**@Min** , **@Max** , ...) sur la classe du "DTO" (ex : Devise) , le statut HTTP alors automatiquement remonté dans l'entête de la réponse HTTP est **400 (Bad Request)** et le corps de la réponse comporte tous les détails sur les éléments invalides .

```
... import jakarta.validation.Valid; ...
public ResponseEntity<Void> ajouterDevise(@Valid @RequestBody Devise devise) { ... }
```

```
...
import jakarta.validation.constraints.Min;
import org.hibernate.validator.constraints.Length;

public class Devise{...
@Length(min=2, max=30, message = "Invalid nom, length should be in [2,30]")
private String nom;

@Min(value=0 ,message = "Invalid change , should be >= 0" )
private String change;
}
```

```
...
@ControllerAdvice
public class RestResponseEntityExceptionHandler
    extends ResponseEntityExceptionHandler {
    ...
    @Override //if @Valid error
    protected ResponseEntity<Object>
        handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
            HttpHeaders headers, HttpStatusCode status, WebRequest request) {
        //return super.handleMethodArgumentNotValid(ex, headers, status, request);
        List<String> errors = ex.getBindingResult()
            .getFieldErrors().stream()
            .map( (FieldError fe) -> fe.getDefaultMessage())
            .collect(Collectors.toList());
        String errorMsg = "not @Valid argument : " + errors.toString();
        return buildResponseEntity(new ApiError(
            HttpStatus.BAD_REQUEST, errorMsg));
    }
    ...
}
```

Résultat en cas de données en entrée incorrectes:

400	{ "status": "BAD_REQUEST", "timestamp": "2024-12-11 01:36:37", "message": "not @Valid argument : [Invalid nom , length should be in [2,30], Invalid change , should be >= 0]"} content-type: application/json
------------	--

3.8. Exemples d'appels en js/ajax

js/ajax-util.js

```
//subfunction with errCallback as optional callback :
function registerCallbacks(xhr, callback, errCallback) {
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            if ((xhr.status == 200 || xhr.status == 204 || xhr.status == 201)) {
                callback(xhr.responseText);
            }
            else {
                if (errCallback)
                    errCallback(withDefaultErrorMessage(xhr,xhr.responseText));
            }
        }
    };
}

function withDefaultErrorMessage(xhr,errorMessage){
    if(errorMessage==null || errorMessage==""){
        switch(xhr.status){
            case 401:
                return "401/Unauthorized (no authentication (ex: no token))";
            case 403:
                return "403/Forbidden (not enough access)";
            case 500:
                return "500/Internal Server Error";
        }
    }
    else return errorMessage;
}

function setTokenInRequestHeader(xhr){
    let authToken = sessionStorage.getItem("authToken");
    if(authToken!=null && authToken!="")
        xhr.setRequestHeader("Authorization", "Bearer " + authToken);
}

function makeAjaxGetRequest(url, callback, errCallback) {
    var xhr = new XMLHttpRequest();
    registerCallbacks(xhr, callback, errCallback);
    xhr.open("GET", url, true);
    setTokenInRequestHeader(xhr);
    xhr.send(null);
}

function makeAjaxDeleteRequest(url, callback, errCallback) {
    var xhr = new XMLHttpRequest();
    registerCallbacks(xhr, callback, errCallback);
    xhr.open("DELETE", url, true);
    setTokenInRequestHeader(xhr);
    xhr.send(null);
}
```

```

function makeAjaxPostRequest(url, jsonData, callback, errCallback) {
    var xhr = new XMLHttpRequest();
    registerCallbacks(xhr, callback, errCallback);
    xhr.open("POST", url, true);
    xhr.setRequestHeader("Content-Type", "application/json");
    setTokenInRequestHeader(xhr);
    xhr.send(jsonData);
}
function makeAjaxPutRequest(url, jsonData, callback, errCallback) {
    var xhr = new XMLHttpRequest();
    registerCallbacks(xhr, callback, errCallback);
    xhr.open("PUT", url, true);
    xhr.setRequestHeader("Content-Type", "application/json");
    setTokenInRequestHeader(xhr);
    xhr.send(jsonData);
}

```

username :	admin1
password :	pwdadmin1
roles :	admin
login	

login successful with roles=admin

login.html

```

<html>
<head> <title>login</title><script src="js/ajax-util.js"></script> <script src="js/login.js"></script>
</head>
<body>
    <h3> login (ws security) </h3>
    username : <input id="txtUsername" type='text' value="admin1"/><br/>
    password : <input id="txtPassword" type='text' value="pwdadmin1"/><br/>
    roles : <input id="txtRoles" type='text' value="admin"/><br/>
    <input type='button' value="login" id="btnLogin"/> <br/>
    <span id="spanMsg"></span> <br/>
    <hr/> <a href="index.html">retour vers index.html</a>
</body>
</html>

```

js/login.js

```

window.onload=function(){
    var spanMsg = document.querySelector('#spanMsg');
    var btnLogin=document.querySelector('#btnLogin');
}

```

```
btnLogin.addEventListener("click" , function (){

    var auth = { username : null, password : null , roles : null } ;
    auth.username = document.querySelector('#txtUsername').value;
    auth.password = document.querySelector('#txtPassword').value;
    auth.roles = document.querySelector('#txtRoles').value;

    var cbLogin = function(data,xhr){
        console.log(data); //data as json string;
        var authResponse = JSON.parse(data);
        if(authResponse.status){
            spanMsg.innerHTML=authResponse.message + " with roles=" + authResponse.roles;
            //localStorage.setItem("authToken",authResponse.token);
            sessionStorage.setItem("authToken",authResponse.token);
        }else{
            spanMsg.innerHTML=authResponse.message ;
        }
    }//end of cbLogin

    var cbError = function(xhr){
        spanMsg.innerHTML= xhrStatusToErrorMessage(xhr) ;
    }

    var xhr = initXhrWithCallback(cbLogin,cbError);
    makeAjaxPostRequest(xhr,"./api-rest/login-api/public/auth" , JSON.stringify(auth));

});//end of btnLogin.addEventListener/click
};//end of window.onload
```

recherche devises selon taux mini (public)

changeMini :

- Euro , 1
- Dollar , 1.1243
- Yen , 121.6477

ajout de monnaie (after logging as ADMIN)

codeMonnaie: (ex: EUR,USD,...)
nommonnaie: (ex: euro,dollar,...)
tauxChange: (ex: 1, 0.85 , 1.5, ...)

{ "code": "ms", "name": "monnaieSinge", "change": 1.23456 }

appel_ajax.html

```
<html>
<head>
    <script src="js/ajax-util.js"></script>    <script src="js/appelAjax.js"></script>
    <meta charset="UTF-8"> <title>appel_ajax</title>
</head>
<body>
    <h3>recherche devises selon taux mini (public)</h3>
    changeMini : <input type="text" id="txtChangeMini" value="1"/> <br/>
        <input type="button" value="getDevises" id="btnGetDevises" /> <br/>
    <div id="divRes"></div>

    <h3> ajout de monnaie (after logging as ADMIN)</h3>
    codeMonnaie: <input type="text" id="txtCode" value="ms" /> (ex: EUR,USD,...)<br/>
    nommonnaie: <input type="text" id="txtName" value="monnaieSinge" /> (ex: euro,dollar,...)<br/>
    tauxChange: <input type="text" id="txtChange" value="1.23456" /> (ex: 1, 0.85 , 1.5, ... )<br/>
    <input type="button" id="btnPostDevise" value="sauvegarder devise" /> <br/>
    <div id="divMessage"></div>
    <hr/>
    <a href="index.html">retour index.html</a>
</body>
</html>
```

js/appelAjx.js

```

window.onload=function(){
    var inputChangeMini = document.querySelector("#txtChangeMini");
    var btnGetDevises = document.querySelector("#btnGetDevises");
    var btnPostDevise = document.querySelector("#btnPostDevise");
    var divRes = document.querySelector("#divRes");
    var divMessage = document.querySelector("#divMessage");
    var cbError = function(xhr){
        divMessage.innerHTML= xhrStatusToErrorMessage(xhr) ;
    }
    btnGetDevises.addEventListener("click" , function (){
        var changeMini = inputChangeMini.value;
        var cbAffDevises=function(texteReponse,xhr){
            //divRes.innerHTML = texteReponse;
            var listeDeviseJs = JSON.parse(texteReponse /* au format json string */)
            var htmlListeDevises = "<ul>" ;
            for(i=0; i<listeDeviseJs.length ; i++){
                htmlListeDevises = htmlListeDevises + "<li>" + listeDeviseJs[i].name + " , "
                + listeDeviseJs[i].change + "</li>";
            }
            htmlListeDevises = htmlListeDevises + "</ul>";
            divRes.innerHTML= htmlListeDevises;
        }
        var xhr = initXhrWithCallback(cbAffDevises , cbError);
        makeAjaxGetRequest(xhr,"./api-rest/devise-api/public/devise?changeMini="+changeMini );
    });//end of btnGetDevises.addEventListener/"click"

    btnPostDevise.addEventListener("click" , function (){
        var nouvelleDevise = { code : null, name : null, change : null };
        nouvelleDevise.code = document.querySelector("#txtCode").value;
        nouvelleDevise.name = document.querySelector("#txtName").value;
        nouvelleDevise.change = document.querySelector("#txtChange").value;
        var cbGererResultatPostDevise = function (texteReponse,xhr){
            divMessage.innerHTML= texteReponse;
        }
        var xhr = initXhrWithCallback(cbGererResultatPostDevise, cbError);
        makeAjaxPostRequest(xhr,"./api-rest/devise-api/private/role_admin/devise" ,
            JSON.stringify(nouvelleDevise));
    });//end of btnGetDevises.addEventListener/"click"
} //end of window.onload

```

3.9. Différentes API pour appeler des WS REST via Spring

Plein de façons pour appeler un WS_REST par code java/spring:

RestTemplate (Spring)	De l'époque java 1.8 (anciennement deprecated puis restauré)
WebClient (Spring)	Api "spécifique Spring_web_flux" : asynchrone/réactive mais complexe et pas portable
HttpRequest/HttpClient (java>=9)	Api standard du langage java (d'un peu plus bas niveau) mais fonctionnant aussi bien en mode synchrone qu'asynchrone (avec CompletableFuture).
RestClient (Spring ≥ 6.1)	Api moderne synchrone s'appuyant sur le pattern "builder"

3.10. Invocation java de service REST via RestTemplate de Spring

Utile pour une **délégation de service** ou bien pour un **test d'intégration** (automatisable via maven et intégration continue).

```
.....  
import org.junit.Assert;  
import org.junit.BeforeClass;  
import org.junit.Test;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.web.client.RestTemplate;  
  
/* cette classe à un nom qui commence ou se termine par IT (et par par Test)  
* car c'est un Test d'Integration qui ne fonctionne que lorsque toute l'application  
* est entièrement démarrée (avec EmbeddedTomcat ou équivalent) . */  
public class PersonWsRestIT {  
  
    private static Logger logger = LoggerFactory.getLogger(PersonWsRestIT.class);  
  
    private static RestTemplate restTemplate; //objet technique de Spring pour test WS REST  
  
    //pas de @Autowired ni de @RunWith  
    //car ce test EXTERNE est censé tester le WebService sans connaître sa structure interne  
    // (test BOITE_NOIRE)  
    @BeforeClass  
    public static void init(){
```

```

restTemplate = new RestTemplate();
}

@Test
public void testGetSpectacleById(){
    final String BASE_URL =
        "http://localhost:8888/spring-boot-spectacle-ws/spectacle-api/public";
    final String uri = BASE_URL + "/spectacle/1";
    String resultAsString = restTemplate.getForObject(uri, String.class);
    logger.info("json string of spectacle 1 via rest: " + resultAsString);
    Spectacle s1 = restTemplate.getForObject(uri, Spectacle.class);
    logger.info("spectacle 1 via rest: " + s1);
    Assert.assertTrue(s1.getId()==1L);
}

@Test
public void testListeComptesDuClient(){
    final String villeDepart = "Paris";
    final String dateDepart = "2018-09-20";
    final String uri = "http://localhost:8080/flight_web/mvc/rest/vols/byCriteria"
        +"?villeDepart=" + villeDepart + "&dateDepart=" + dateDepart;
    String resultAsString = restTemplate.getForObject(uri, String.class);
    logger.info("json listeVols via rest: " + resultAsString);
    Vol[] tabVols = restTemplate.getForObject(uri, Vol[].class);
    logger.info("java listeComptes via rest: " +tabVols.toString());
    Assert.assertNotNull(tabVols); Assert.assertTrue(tabVols.length>=0);
    for(Vol cpt : tabVols){
        System.out.println("\t" + cpt.toString());
    }
}

@Test
public void testVirement(){
    final String uri =
        "http://localhost:8080/tpSpringWeb/mvc/rest/compte/virement";
    //post/envoi:
    OrdreVirement ordreVirement = new OrdreVirement();
}

```

```

        ordreVirement.setMontant(50.0);
        ordreVirement.setNumCptDeb(1L);
        ordreVirement.setNumCptCred(2L);
        OrdreVirement savedOrdreVirement =
            restTemplate.postForObject(uri, ordreVirement, OrdreVirement.class);
        logger.info("savedOrdreVirement via rest: " + savedOrdreVirement.toString());
        Assert.assertTrue(savedOrdreVirement.getOk().equals(true));
    }
}

```

Exemple 2 (délégation de service) :

```

...
import java.nio.charset.Charset;
import java.util.Base64;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.util.LinkedMultiValueMap;
import org.springframework.util.MultiValueMap;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
@RequestMapping(value="/myapi/auth" ,  headers="Accept=application/json")
public class LoginDelegateCtrl {

    private static Logger logger = LoggerFactory.getLogger(LoginDelegateCtrl.class);
}

```

```

private static final String ACCESS_TOKEN_URL =
    "http://localhost:8081/basic-oauth-server/oauth/token";

private static RestTemplate restTemplate = new RestTemplate();

HttpHeaders createBasicHttpAuthHeaders(String username, String password){
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
    String auth = username + ":" + password;
    byte[] encodedAuth = Base64.getEncoder().encode(
        auth.getBytes(Charset.forName("US-ASCII")));
    String authHeader = "Basic " + new String( encodedAuth );
    headers.add("Authorization", authHeader);
    return headers;
}

@PostMapping("/login")
public ResponseEntity<?> authenticateUser(@RequestBody AuthRequest loginRequest) {
    logger.debug("/login , loginRequest:"+loginRequest);
    String authResponse="{}";
    try{
        MultiValueMap<String, String> params= new LinkedMultiValueMap<String,
String>();
        params.add("username", loginRequest.getUsername());
        params.add("password", loginRequest.getPassword());
        params.add("grant_type", "password");
        //ResponseEntity<String> tokenResponse =
        //    restTemplate.postForEntity(ACCESS_TOKEN_URL,params, String.class);
        // si pas besoin de spécifier headers spécifique .
    }
    HttpHeaders headers = createBasicHttpAuthHeaders("fooClientIdPassword","secret");
    HttpEntity<MultiValueMap<String, String>> entityReq =
        new HttpEntity<MultiValueMap<String, String>>(params, headers);

    ResponseEntity<String> tokenResponse=
        restTemplate.exchange(ACCESS_TOKEN_URL,
            HttpMethod.POST,

```

```

    entityReq,
    String.class);

authResponse=tokenResponse.getBody();

logger.debug("/login authResponse:" + authResponse.toString());

return ResponseEntity.ok(authResponse);

}

catch (Exception e) {

    logger.debug("echec authentification:" + e.getMessage()); //for log
    return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
        .body(authResponse);

}

}

}

```

3.11. Appel moderne/asynchrone de WS-REST avec WebClient

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    <!-- pour appel de WS-REST externes , WebClient mieux que RestTemplate -->
</dependency>

```

RestClientApp.java

```

package tp.appliSpring.client;

import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;
import tp.appliSpring.dto.Currency; import tp.appliSpring.dto.LoginRequest;
import tp.appliSpring.dto.LoginResponse;

public class RestClientApp {

public static String token="?";

public static void main(String[] args) {
    postLoginForToken();
    posterNouvelleDevise();
}

private static void postLoginForToken() {
    WebClient.Builder builder = WebClient.builder();
    String baseUrl="http://localhost:8080/appliSpring/api-bank";
    WebClient webClient = builder

```

```

.baseUrl(baseUrl)
.defaultHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
.build();

LoginRequest loginRequest = new LoginRequest("admin1","pwd1");

//envoyer cela via un appel en POST
Mono<LoginResponse> reactiveStream = webClient.post().uri("/public/login")
    .body(Mono.just(loginRequest), LoginRequest.class)
    .retrieve()
    .bodyToMono(LoginResponse.class)
    .onErrorReturn(new LoginResponse("admin1",false,"login failed",null));
LoginResponse loginResponse = reactiveStream.block();

System.out.println("loginResponse=" + loginResponse.toString());
if(loginResponse.getOk())
    token = loginResponse.getToken();
}

private static void posterNouvelleDevise() {
    WebClient.Builder builder = WebClient.builder();
    String baseUrl="http://localhost:8080/appliSpring/api-bank";
    WebClient webClient = builder
        .baseUrl(baseUrl)
        .defaultHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
        .defaultHeader(HttpHeaders.AUTHORIZATION, "Bearer " + token)
        .build();

    //créer une instance du DTO Currency
    //avec les valeurs
    //{{ "code" : "DDK" , "name" : "couronne danoise" , "rate" : 7.77 }

    Currency currencyDDK = new Currency("DDK","couronne danoise" , 7.77);

    //envoyer cela via un appel en POST
    Mono<Currency> reactiveStream = webClient.post().uri("/devise")
        .body(Mono.just(currencyDDK), Currency.class)
        .retrieve()
        .bodyToMono(Currency.class)
        .onErrorReturn(new Currency("?", "not saved !!",0.0));

    Currency savedCurrency = reactiveStream.block();

    System.out.println("savedCurrency=" + savedCurrency.toString());
}
}

```

Variantes pour appel(s) en mode GET :

```

private String tempApiKey="26ca93ee7fc19cbe0a423aaa27cab235";
private String fixerApiUrl="http://data.fixer.io/api/latest"
    +"?access_key="+tempApiKey; //apiKey may be passed in header with other api
/*

```

```

Mono<String> reactiveStream = webClient.get()
    .retrieve()
    .bodyToMono(new ParameterizedTypeReference<String>() {});
String result = reactiveStream.block();
System.out.println("result=" + result);
*/
//type de réponse brute attendue:
/*
{"success":true,"timestamp":1635959583,"base":"EUR","date":"2021-11-03",
"rates":{"AED":4.254663,"AFN":105.467869,..., "EUR":1 , ...}}
*/
Mono<FixerIoResponse> reactiveStream = webClient.get() //uri("/suiteUrlQuiVaBien")
    .retrieve()
    .bodyToMono(new ParameterizedTypeReference<FixerIoResponse>() {});
FixerIoResponse fixerIoResponse = reactiveStream.block();

/*
    ResponseEntity<FixerIoResponse> fixerIoResponseEntity=
        webClient.get().retrieve()
        .toEntity(FixerIoResponse.class).block();

    FixerIoResponse fixerIoResponse = null;
    if(fixerIoResponseEntity.getStatusCode()==HttpStatus.OK) {
        fixerIoResponse=fixerIoResponseEntity.getBody();
    }
*/

```

3.12. Invocation via l'api standard HTTP2 de java ≥ 9

MyHttp2Util.java

```

package tp.appliSpring.http2;

import java.net.InetSocketAddress;
import java.net.ProxySelector;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.net.http.HttpResponse.BodyHandlers;
import java.util.List;

import com.fasterxml.jackson.databind.DeserializationFeature;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.type.CollectionType;

public class MyHttp2Util {
    public static MyHttp2Util INSTANCE = new MyHttp2Util();

    private HttpClient client = HttpClient.newBuilder()
        //.proxy(ProxySelector.of(new InetSocketAddress("100.78.112.201", 8001)))
        .build();
}

```

```

private ObjectMapper jsonMapper = new ObjectMapper();

public MyHttp2Util() {jsonMapper.configure(
    DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
}

public String fetchAsJsonString(String url){
    String jsonString=null;
    try {
        HttpRequest req =
            HttpRequest.newBuilder(URI.create(url))
            .header("User-Agent", "Java")
            .GET()
            .build();
        HttpResponse<String> resp =
            client.send(req, BodyHandlers.ofString());
        if(resp.statusCode()==200) {
            jsonString=resp.body();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return jsonString;
}

public <T> T fetch(String url,Class<T> dataClass){
    T result=null;
    try {
        String jsonString=fetchAsJsonString(url);
        if(jsonString!=null)
            result=jsonMapper.readValue(jsonString, dataClass);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return result;
}

public <T> List<T> fetchList(String url,Class<T> dataClass){
    List<T> result=null;
    try {
        String jsonString=fetchAsJsonString(url);
        CollectionType javaType = jsonMapper.getTypeFactory()
            .constructCollectionType(List.class, dataClass);
        if(jsonString!=null)
            result=jsonMapper.readValue(jsonString, javaType);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return result;
}
}

```

Utilisation :

```
public class MoneyExchangeRate {

    public record Response(Boolean success, Long timestamp, String base, String date,
                           HashMap<String, Double> rates) {
    }

    public static String buildCurrentRatesURL(String apiKey) {
        return "http://data.fixer.io/api/latest"
            + "?access_key=" + apiKey;
    }

    public static String buildCurrentRatesURL() {
        String defaultApiKey = "26ca93ee7fc19cbe0a423aaa27cab235"; // didierDefrance
        return buildCurrentRatesURL(defaultApiKey);
    }

}
```

```
...
@Test
public void testMoneyExchangeRate() {
    MoneyExchangeRate.Response moneyExchangeRates =
        myHttp2Util.fetch(MoneyExchangeRate.buildCurrentRatesURL(),
                           MoneyExchangeRate.Response.class);
    assertNotNull(moneyExchangeRates);
    logger.debug("moneyExchangeRates=" + moneyExchangeRates);
    logger.debug("1 euro=" + moneyExchangeRates.rates().get("USD") + "USD");
}
```

Résultats :

moneyExchangeRates=Response[success=true, timestamp=1733924956, base=EUR, date=2024-12-11, rates={FJD=2.436811, ..., USD=1.053164, ..., GBP=0.824427,...}]

1 euro=1.053164USD

3.13. Via RestClient de Spring ≥ 6.1

Exemple d'appel élémentaire en mode "GET" :

```
package tp.xyz.ws_call;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.util.LinkedMultiValueMap;
import org.springframework.util.MultiValueMap;
import org.springframework.web.client.RestClient;

public class WsCallWithRestClient {

    public static void testRestClientGetZippopotam(){
        RestClient restClient = RestClient.create();
        String uriBase = "http://api.zippopotam.us";

        String resultAsString = restClient.get()
            .uri(uriBase + "/fr/75001")
            .retrieve()
            .body(String.class);
        System.out.println("resultAsString=" + resultAsString);

        ZippopotamResponse resultAsObject = restClient.get()
            .uri(uriBase + "/fr/75001")
            .retrieve()
            .body(ZippopotamResponse.class);
        System.out.println("resultAsObject=" + resultAsObject);
    }
}
```

Récupération du résultat d'une réponse complète avec code de retour :

```
String uriBase = "https://www.d-defrance.fr/tp/devise-api/v1/public";
ResponseEntity<DeviseDto> deviseEurRespEntity = restClient.get()
    .uri(uriBase+ "/devises/EUR")
    .accept(MediaType.APPLICATION_JSON)
    .retrieve()
    .toEntity(DeviseDto.class);

System.out.println("deviseEurRespEntity=" + deviseEurRespEntity.getStatusCode().value());
```

Récupération du résultat sous forme de liste de Dto java :

```
List<DeviseDto> devises = restClient.get()
    .uri(uriBase+ "/devises")
    .accept(MediaType.APPLICATION_JSON)
    .retrieve()
    .body(new ParameterizedTypeReference<List<DeviseDto>>(){});
System.out.println("devises="+devises);
```

Exemple d'appel en mode POST :

```
public static void testRestClientPostDevise(){
    RestClient restClient = RestClient.create();
    String uriBase = "https://www.d-defrance.fr/tp/devise-api/v1/public";

    DeviseDto newDevise = new DeviseDto("Xxx","DeviseXxx",1.123456789);
    //ResponseEntity<Void> if .toBodilessEntity();
    ResponseEntity<DeviseDto> postDeviseRespEntity = restClient.post()
        .uri(uriBase + "/devises")
        .contentType(MediaType.APPLICATION_JSON)
        .body(newDevise)
        .retrieve()
        .toEntity(DeviseDto.class);

    System.out.println("postDeviseRespEntity="+postDeviseRespEntity);
    System.out.println("postDeviseRespEntity.status="+postDeviseRespEntity.getStatusCode().value());
    System.out.println("postDeviseRespEntity.body="+postDeviseRespEntity.getBody());
    System.out.println("postDeviseRespEntity.location_with_id="
        +postDeviseRespEntity.getHeaders().get("Location").get(0));
}
```

Exemple d'appel en mode DELETE :

```
public static void testRestClientDeleteDevise(){
    RestClient restClient = RestClient.create();
    String uriBase = "https://www.d-defrance.fr/tp/devise-api/v1/public";
    try {
        ResponseEntity<Void> deleteDeviseResponseEntity = restClient.delete()
            .uri(uriBase + "/devises/Xxx")
            .retrieve()
            .toBodilessEntity();

        System.out.println("deleteDeviseResponseEntity.status="
            +deleteDeviseResponseEntity.getStatusCode().value());
    } catch (Exception e) {
        System.err.println("echec delete devise: " + e.getMessage());
    }
}
```

Exemple d'appel avec sécurité OAuth2 où le mode grant_type=password est autorisé :

```
public static String fetchToken(){
    //fetch keycloak token via a MediaType.APPLICATION_FORM_URLENCODED request:

    RestClient restClient = RestClient.create();
    String uriBase = "https://www.d-defrance.fr/keycloak/realms/sandboxrealm";
    String clientId = "myspringclient"; // "anywebappclient";
    String clientSecret = "crtwAVOIGxmjXRrosDYxEEKlPCMe18rp";

    MultiValueMap<String, String> formParamMap= new LinkedMultiValueMap<>();
    formParamMap.add("username", "admin1");
    formParamMap.add("password", "pwd1");
    formParamMap.add("client_id", clientId);
    formParamMap.add("client_secret", clientSecret);
    formParamMap.add("grant_type", "password");

    ResponseEntity<Map> responseFetchToken = restClient.post()
        .uri(uriBase + "/protocol/openid-connect/token")
        .body(formParamMap)
        .contentType(MediaType.APPLICATION_FORM_URLENCODED)
        .retrieve()
        .toEntity(Map.class);

    System.out.println("responseFetchToken.status="+responseFetchToken.getStatusCode().value());
    System.out.println("responseFetchToken.body as map="+responseFetchToken.getBody());
    return (String) responseFetchToken.getBody().get("access_token");
}
```

```
public static void testRestClientPutCompteWithSecurity(){
    RestClient restClient = RestClient.create();
    String uriBase = "http://localhost:8181/appliSpring/rest/api-bank/v1";

    String token =fetchToken();
    System.out.println("oauth2 token=" + token);

    Map<String, Object> compteDtoAsMap = new HashMap<>();
    compteDtoAsMap.put("numero", 1L);
    compteDtoAsMap.put("label", "new label for compte 3");
    compteDtoAsMap.put("solde", 1234.5);
    ResponseEntity<Map> putCompteRespEntity = restClient.put()
        .uri(uriBase + "/comptes/3")
        .contentType(MediaType.APPLICATION_JSON)
        .header("Authorization", "Bearer " + token)
        .body(compteDtoAsMap)
        .retrieve()
        .toEntity(Map.class);
    System.out.println("putCompteRespEntity=" + putCompteRespEntity);
    System.out.println("putCompteRespEntity.status=" + putCompteRespEntity.getStatusCode().value());
}
```

3.14. Test d'un "RestController" via MockMvc

Pour tester le comportement d'un composant "RestController" de Spring-Mvc sans avoir à préalablement démarrer l'application complète, on peut utiliser la classe **MockMvc** et l'annotation **@WebMvcTest** ou bien **@AutoConfigureMockMvc** qui sont spécialement prévues pour faire fonctionner le code d'un web service rest de spring-mvc en recréant un contexte local ayant à peu près de même comportement que celui d'un conteneur web mais sans accès réseau/http .

Deux Grandes Variantes :

- via **@WebMvcTest** : test unitaire avec mock de service interne
- via **@SpringBootTest** et **@AutoConfigureMockMvc** : test d'intégration avec réels services

Tests de @RestController

éventuels tests via PostMan ou curl ou Swagger/OpenApi en mode "try-out"

```
@SpringBootTest
@AutoConfigureMockMvc
TestXyzCtrlWithRealService{
    @Autowired MockMvc mvc ;
    ...
    mvc.perform(get("..."))
        .andExpect(...)... ;
}
```

```
@WebMvcTest(XyzCtrl.class)
TestXyzCtrlWithServiceMock {
    @Autowired MockMvc mvc ;
    @MockBean XyzService xyzSMock ;
    ...
    Mockito.when(...).thenReturn(...);
    mvc.perform(get("..."))
        .andExpect(...)... ;
}
```

```
@RestController
@RequestMapping(...)
class XyzCtrl{
    @GetMapping(...)
    Xyz getById(...){
        return xyzService.searchById(id) ;
    }
    ...
}
```

xyzSMock

XyzService

XyzRepository



3.15. Test unitaire de contrôleur Rest

```
package tp.appliSpring.rest;

import static org.hamcrest.Matchers.hasSize;
import static org.hamcrest.Matchers.is;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
```

```

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import java.util.ArrayList;           import java.util.List;
import org.junit.jupiter.api.BeforeEach;  import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;  import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;
import tp.appliSpring.entity.Compte;  import tp.appliSpring.service.CompteService;

@ExtendWith(SpringExtension.class) //si junit5/jupiter
@WebMvcTest(CompteRestCtrl.class)
//NB: @WebMvcTest without security and without service layer , service must be mocked !!!
public class TestCompteRestCtrlWithServiceMock {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private CompteService compteService; //not real implementation but mock to configure .

    @BeforeEach
    public void reInitMock() {
        //vérification que le service injecté est bien un mock
        assertTrue(Mockito.mockingDetails(compteService).isMock());
        //reinitialisation du mock(de scope=Singleton par défaut) sur aspects stub et spy
        Mockito.reset(compteService);
    }

    @Test //à lancer sans le profile withSecurity
    public void testComptesDuClient1WithMockOfCompteService(){

        //préparation du mock (qui sera utilisé en arrière plan du contrôleur rest à tester):
        List<Compte> comptes = new ArrayList<>();
        comptes.add(new Compte(1L,"compteA",40.0));
        comptes.add(new Compte(2L,"compteB",90.0));
        Mockito.when(compteService.comptesDuClient(1)).thenReturn(comptes);

        try {
            MvcResult mvcResult =
                mvc.perform(get("/api-bank/compte?numClient=1")
                    .contentType(MediaType.APPLICATION_JSON))
                    .andExpect(status().isOk())
                    .andExpect(jsonPath("$", hasSize(2)))
                    .andExpect(jsonPath("$.label", is("compteA")))
                    .andExpect(jsonPath("$.solde", is(90.0)))
                    .andReturn();
            System.out.println(">>>>>>> jsonResult=" +
                mvcResult.getResponse().getContentAsString());
        }
    }
}

```

```
    } catch (Exception e) {
        System.err.println(e.getMessage());
    }
}
```

NB : Spring5 propose une variante `@WebFluxTest` et `WebTestClient` pour `WebFlux` .

3.16. Test d'intégration de contrôleur Rest avec réels services

```

package tp.appliSpring.rest;
import static org.hamcrest.Matchers.hasSize;
import static org.hamcrest.Matchers.is;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;

@ExtendWith(SpringExtension.class) //si junit5/jupiter
@SpringBootTest //with all layers
@AutoConfigureMockMvc //to test controller with real spring services implementations
@ActiveProfiles({"embededDb","init"}) //init profile for ...init.ReinitDefaultDataSet
public class TestCompteRestCtrlWithRealService {

    @Autowired
    private MockMvc mvc;

    @Test //à lancer sans le profile withSecurity
    public void testComptesDuClient1WithRealService(){
        try {
            MvcResult mvcResult =
                mvc.perform(get("/api-bank/compte?numClient=1")
                    .contentType(MediaType.APPLICATION_JSON))
                    .andExpect(status().isOk())
                    .andExpect(jsonPath("$.label", hasSize(2)))
                    .andExpect(jsonPath("$[0].label", is("compteA")))
                    .andReturn();
            //à adapter selon jeux de données de init.ReInitDefaultDataset
            System.out.println(">>>>>>> jsonResult=" +
                mvcResult.getResponse().getContentAsString());
        } catch (Exception e) {
            System.err.println(e.getMessage());
            //e.printStackTrace();
        }
    }
}

```

4. Config swagger3 / openapi-doc pour spring

Ancienne version à ne pas ajouter dans pom.xml

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

Version plus récente à ajouter dans pom.xml

```
<!-- springdoc-openapi-ui for spring5/springBoot2 ,
     springdoc-openapi-starter-webmvc-ui for spring6/springBoot3 -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.3.0</version>
</dependency>
```

en plus de

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

Configuration explicite à idéalement placer dans application.properties :

```
springdoc.swagger-ui.path=/doc-swagger.html
```

dans index.html (ou ailleurs) :

```
<a href="/doc-swagger.html">documentation Api REST générée dynamiquement par swagger3/openapi</a>
```

The screenshot shows the Swagger UI interface for the 'api-bank' API, which is version v1 and follows OAS 3.0. The URL in the browser is `localhost:8181/appliSpring/swagger-ui/index.html#/`. The main title is 'api-bank'. Below it, there's a link to '/appliSpring/v3/api-docs'. The page includes a logo for Swagger supported by SMARTBEAR, a 'minibank api (tp)' link, and a 'GNU/GPL' link. A 'Servers' section shows the URL `http://localhost:8181/appliSpring - Generated server url`.

compte-rest-ctrl

GET /rest/api-bank/v1/comptes/{id}

PUT /rest/api-bank/v1/comptes/{id}

DELETE /rest/api-bank/v1/comptes/{id}

GET /rest/api-bank/v1/comptes

POST /rest/api-bank/v1/comptes post a new account to be created

NB : Selon le contexte applicatif , il faudra peut être paramétrer la sécurité de façon à pouvoir accéder à la documentation "swagger" générée :

```

@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter
{
    //....
    private static final String[] SWAGGER_AUTH_WHITELIST = {
        "/swagger-resources/**",  "/swagger-ui.html",  "/v2/api-docs",   "/webjars/**"
    };

    protected void configure(HttpSecurity http) throws Exception {
        // configuration partielle à compléter:
        http.authorizeRequests()
            .antMatchers("/rest/devise-api/public/**").permitAll()
            .antMatchers(SWAGGER_AUTH_WHITELIST).permitAll()
            .anyRequest().authenticated();
    }
}

```

Configuration de l'api via "annotations OpenApi/ swagger 3" :

Attention : les anciennes annotations de l'époque "swagger2" (**@ApiModelProperty** , **@ApiOperation** , **@ApiParam**) ont été changées lors de la standardisation swagger3/OpenApi (**@Schema** , **@Operation** , **@Parameter** , ...)

```

package org.mycontrib.backend.dto;
import io.swagger.v3.oas.annotations.media.Schema;
@Schema(description = "DTO (Result of conversion)")
public class ResConv {
    @Schema(description = "amount to convert", defaultValue = "100")
    private Double amount;

    @Schema( description = "source currency code", defaultValue = "EUR")
    private String source;
    ... // @Schema(hidden = true) ou bien @JsonIgnore ou ...
}

```

et dans une classe de @RestController :

```
...
import io.swagger.v3.oas.annotations.media.Operation;
import io.swagger.v3.oas.annotations.media.Parameter;
...
@RestController
@RequestMapping(value="/rest/devise-api/public" , headers="Accept=application/json")
public class PublicDeviseRestCtrl {
...
@RequestMapping(value="/convert" , method=RequestMethod.GET)
@Operation(summary= "convert amount from source to target currency",
    description = "exemple: convert?source=EUR&target=USD&amount=100")
    public ResConv convertir(
        @RequestParam("amount")
        @Parameter(description = "amount to convert", ... = "100")
        Double montant,
        @RequestParam("source")
        @Parameter(description = "source currency code", ... = "EUR")
        String source,
        @RequestParam("target")
        @Parameter(description = "target currency code", ... = "USD")
        String cible) {
        Double res = convertisseur.convertir(montant, source, cible);
        return new ResConv(montant, source, cible,res);
    }
....
```

Précision sur les codes et messages possibles en retour :

```
import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.media.Content;
import io.swagger.v3.oas.annotations.media.Schema;
import io.swagger.v3.oas.annotations.responses.ApiResponse;
import io.swagger.v3.oas.annotations.responses.ApiResponses;
...
@Operation(summary= "post a new account to be created")
@ApiResponses({
    @ApiResponse(responseCode = "201" , , description = "Created"
        content = {@Content(mediaType = "application/json",
            schema = @Schema(implementation = Compte.class ))}),
    @ApiResponse(responseCode = "500" , description = "Internal Server Error",
        content = {@Content(schema = @Schema(implementation = ApiError.class)))} })
@PostMapping("")
public ResponseEntity<?> postCompte(@Valid @RequestBody CompteToCreate compte) {...}
```

Pour simplifier les paramétrages un peu trop verbeux si par annotations seulement , on peut s'appuyer sur un complément de configuration de ce type (ci dessous avec ou sans sécurité):

OpenApiDocConfig.java

```
package tp.appliSpring.bank.web.api;  
import io.swagger.v3.oas.annotations.OpenAPIDefinition;  
import io.swagger.v3.oas.annotations.enums.SecuritySchemeType;  
import io.swagger.v3.oas.models.Components;  
import io.swagger.v3.oas.models.OpenAPI;  
import io.swagger.v3.oas.models.info.Info;  
import io.swagger.v3.oas.models.info.License;  
import io.swagger.v3.oas.models.media.*;  
import io.swagger.v3.oas.models.responses.ApiResponse;  
import io.swagger.v3.oas.models.security.*;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Profile;
```

@Configuration

@OpenAPIDefinition

```
public class OpenApiDocConfig {  
    private static final String OAUTH_SCHEME = "OAuth";  
  
    @Value("${springdoc.oauthFlow.authorizationUrl:'?'})  
    String authorizationUrl;  
  
    @Value("${springdoc.oauthFlow.tokenUrl:'?'})  
    String tokenUrl;  
  
    @Bean  
    @Profile("!withSecurity")  
    public OpenAPI withoutSecurityOpenAPI(Components components) {  
        return this.baseOpenAPI(components);  
    }  
}
```

```

@Bean
@Profile("withSecurity")
public OpenAPI withSecurityOpenAPI(Components components) {
    var openapiConfig = this.baseOpenAPI(components);
    openapiConfig.addSecurityItem(new SecurityRequirement().addList(OAUTH_SCHEME));
    //+ with .addSecuritySchemes(OAUTH_SCHEME, oAuthScheme) in .components()
    return openapiConfig;
}

public OpenAPI baseOpenAPI(Components components) {
    var openapiConfig = new OpenAPI()
        //.servers(servers)
        .info(new Info()
            .title("api-bank")
            .description("minibank api (tp)")
            .version("v1")
            .license(new License().name("GNU/GPL")
                .url("https://www.gnu.org/licenses/gpl-3.0.html"))
        )
        .components(components);
    return openapiConfig;
}

@Bean
@Profile("!withSecurity")
public Components withoutSecurityOpenAPIComponents() {
    return this.baseOpenAPIComponents();
}

@Bean
@Profile("withSecurity")
public Components withSecurityOpenAPIComponents() {
    var oauthFlow = new OAuthFlow()
        .authorizationUrl(this.authorizationUrl)
        .tokenUrl(this.tokenUrl)
        .scopes(new Scopes());
    var oAuthScheme = new SecurityScheme()

```

```

.name(OAUTH_SCHEME)
.type(SecurityScheme.Type.OAUTH2)
.flows(new OAuthFlows().authorizationCode(oauthFlow));

Components components = this.baseOpenAPIComponents();
components.addSecuritySchemes(OAUTH_SCHEME, oAuthScheme);
return components;
}

public Components baseOpenAPIComponents() {
    var noContentResponse = new ApiResponse()
        .description("Sucessfull Operation With NO_CONTENT");
    var compteSchema = new ObjectSchema()
        .name("Compte")
        .title("Compte")
        .description("Bank Account")
        .addProperties("numero", new IntegerSchema().example("1"))
        .addProperties("label", new StringSchema().example("myBankAccount"))
        .addProperties("solde", new NumberSchema().example("50.0"));
    var compteContent = new Content()
        .addMediaType("application/json",
            new MediaType().schema(compteSchema));
    var compteResponse = new ApiResponse()
        .description("Compte").content(compteContent);
    var createdCompteResponse = new ApiResponse()
        .description("Created Compte (with id)").content(compteContent);

    var apiErrorSchema = new ObjectSchema().name("ApiError")
        .title("ApiError").description("ApiError message")
        .addProperties("status", new StringSchema()
            .example("NOT_FOUND or INTERNAL_SERVER_ERROR or ..."))
        .addProperties("message", new StringSchema()
            .example("xyz not found or internal server error or ..."))
        .addProperties("timestamp", new StringSchema().example("2024-11-26 08:45:55"));
}

```

```

var apiErrorContent = new Content()
    .addMediaType("application/json" ,
        new MediaType().schema(apiErrorSchema));
var internalServerErrorResponse = new ApiResponse()
    .description("Internal Server Error").content(apiErrorContent);
var notFoundErrorResponse = new ApiResponse()
    .description("Not Found").content(apiErrorContent);

return new Components()
    .addSchemas("CompteSchema" , compteSchema)
    .addSchemas("ApiErrorSchema" , apiErrorSchema)
    .addResponses("NoContentResponse",noContentResponse)
    .addResponses("InternalServerErrorResponse",internalServerErrorResponse)
    .addResponses("NotFoundErrorResponse",notFoundErrorResponse)
    .addResponses("CompteResponse",compteResponse)
    .addResponses("CreatedCompteResponse",createdCompteResponse);
}
}

```

Et grâce à ceci , via ref = "#/components/responses/XyzResponse" , le paramétrage des annotations @ApiResponse est considérablement simplifié :

```

@ApiResponse(responseCode = "200", ref = "#/components/responses/CompteResponse")
@ApiResponse(responseCode = "404",
            ref = "#/components/responses/NotFoundErrorResponse")
@ApiResponse(responseCode = "500",
            ref = "#/components/responses/InternalServerErrorResponse")
@GetMapping("/{id}")
public Compte getCompteById(@PathVariable("id") long numeroCompte) {
    return serviceCompte.searchById(numeroCompte);
    //NB: l'objet retourné sera automatiquement converti au format json
}

```

Effets de la configuration "openApiDoc" sur la description générée :

Responses	
Code	Description
200	<p>Compte</p> <p>Media type</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">application/json</div> <p>Controls Accept header.</p> <p>Example Value Schema</p> <pre>{ "numero": 1, "label": "myBankAccount", "solde": 50 }</pre>
404	<p>Not Found</p> <p>Media type</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">application/json</div> <p>Example Value Schema</p> <pre>{ "status": "NOT_FOUND or INTERNAL_SERVER_ERROR or ...", "message": "xyz not found or internal server error or ...", "timestamp": "2024-11-26 08:45:55" }</pre>
500	Internal Server Error

Les "default values" sont celles qui sont proposées lors d'un "try-it-out" :

Request body	required
	<pre>{ "label": "myBankAccount", "solde": -999 }</pre>

Comportement de swagger/openApiDoc en mode "springSecurity+OAuth2" :

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=https://www.d-defrance.fr/keycloak/realms/sandboxrealm
spring.mvc.pathmatch.matching-strategy=ANT_PATH_MATCHER
springdoc.swagger-ui.oauth.clientId=anywebappclient
springdoc.swagger-ui.oauth.clientSecret=noNeedOfSecret
springdoc.oauthFlow.authorizationUrl=${spring.security.oauth2.resourceserver.jwt.issuer-uri}/protocol/openid-connect/auth
springdoc.oauthFlow.tokenUrl=${spring.security.oauth2.resourceserver.jwt.issuer-uri}/protocol/openid-connect/token
```



Sans login :

401 Error: response status is 401

Avec Login (click sur bouton "Authorize"):

Available authorizations <p>Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes. API requires the following scopes. Select which ones you want to grant to Swagger UI.</p> <p>OAuth (OAuth2, authorizationCode)</p> <p>Authorization URL: https://www.d-defrance.fr/keycloak/realms/sandboxrealm/protocol/openid-connect/auth Token URL: https://www.d-defrance.fr/keycloak/realms/sandboxrealm/protocol/openid-connect/token Flow: authorizationCode client_id: <input type="text" value="anywebappclient"/> client_secret: <input type="password" value="*****"/> <input type="button" value="Authorize"/> <input type="button" value="Close"/> </p>	SANDBOXREALM <p>Sign in to your account</p> <p>Username or email <input type="text" value="admin1"/></p> <p>Password <input type="password" value="*****"/></p> <p><input type="button" value="Sign In"/></p>
--	---

Logout

Close

puis 204 ou bien

403

Error: response status is 403

selon les droits d'accès de l'utilisateur authentifié .

XI - Spring Security (l'essentiel)

1. Extension Spring-security (généralités)

L'extension **Spring-security** permet de simplifier le paramétrage de la **sécurité JEE** dans le cadre d'une application JEE/Web basée sur Spring.

1.1. Principales fonctionnalités de spring-security

Spring-security (fonctionnalités)

- **Configurer les zones web protégées** (URL publiques et URL nécessitant authentification)
- **Configurer le mode d'authentification** (HttpBasic ou BearerToken , formulaire de login, ...)
- **Configurer un accès à un "realm"** (*liste d'utilisateurs* pouvant s'authentifier (*via username/password*) et ayant des *rôles* et/ou des *permissions/privilèges*) (variantes : InMemory, JDBC, OAuth2/OIDC , UserDetailsService spécifique)
- **Intégration Spring et compatibilité JavaWeb (WebFilter,...)**

Autres caractéristiques de spring-security :

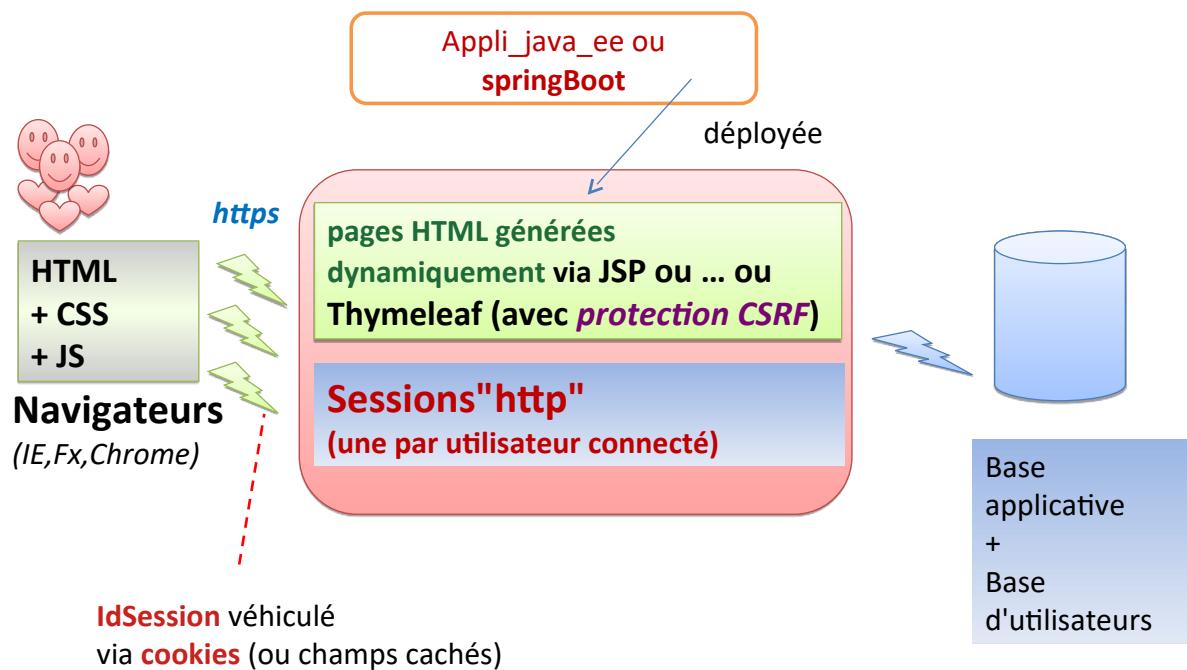
- syntaxe xml ou java simplifiée (plus compacte et plus lisible que le standard "web.xml")
- possibilité de configurer via l'annotation `@PreAuthorize("hasRole('role1')")` les méthodes des composants "spring" qui seront ou pas accessibles selon le rôle de l'utilisateur authentifié.
- cryptage des mots de passe via bcrypt, ...

1.2. Principaux besoins types (**spring-security**)

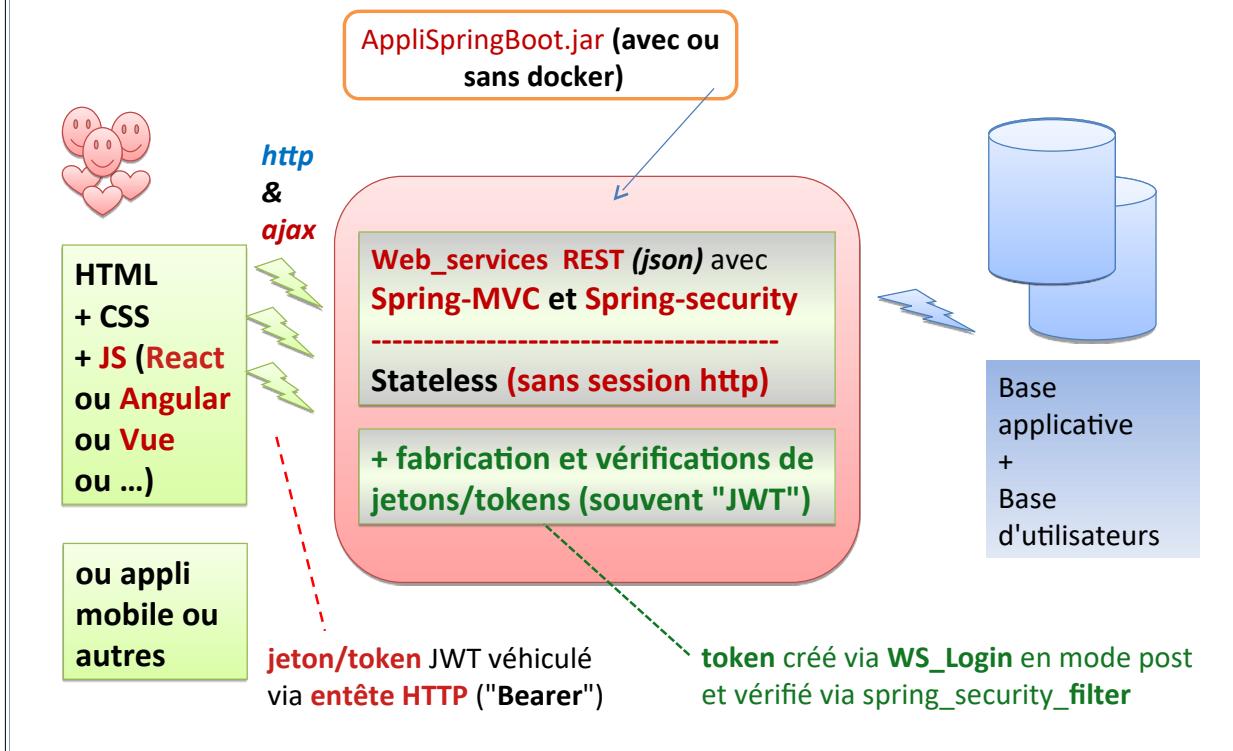
Spring-security (besoins classiques)

- **Partie d'appli Web** (avec **@Controller** et **pages JSP** ou bien **Thymeleaf**) avec sécurité JavaEE classique (*id de HttpSession* véhiculé par *cookie* , authentification Basic Http et formulaire de login)
- **Partie Api REST** (avec **@RestController**) et avec **BearerToken** (ex : **JWT**) gérée par le backend springBoot en mode **standalone**
- **Api REST en mode "ResourceServer"** où **l'authentification est déléguée** via **OAuth2/OIDC** à un **"AuthorizationServer"** (ex : KeyCloak , Cognito , Azure-Directory, Okta , ...)

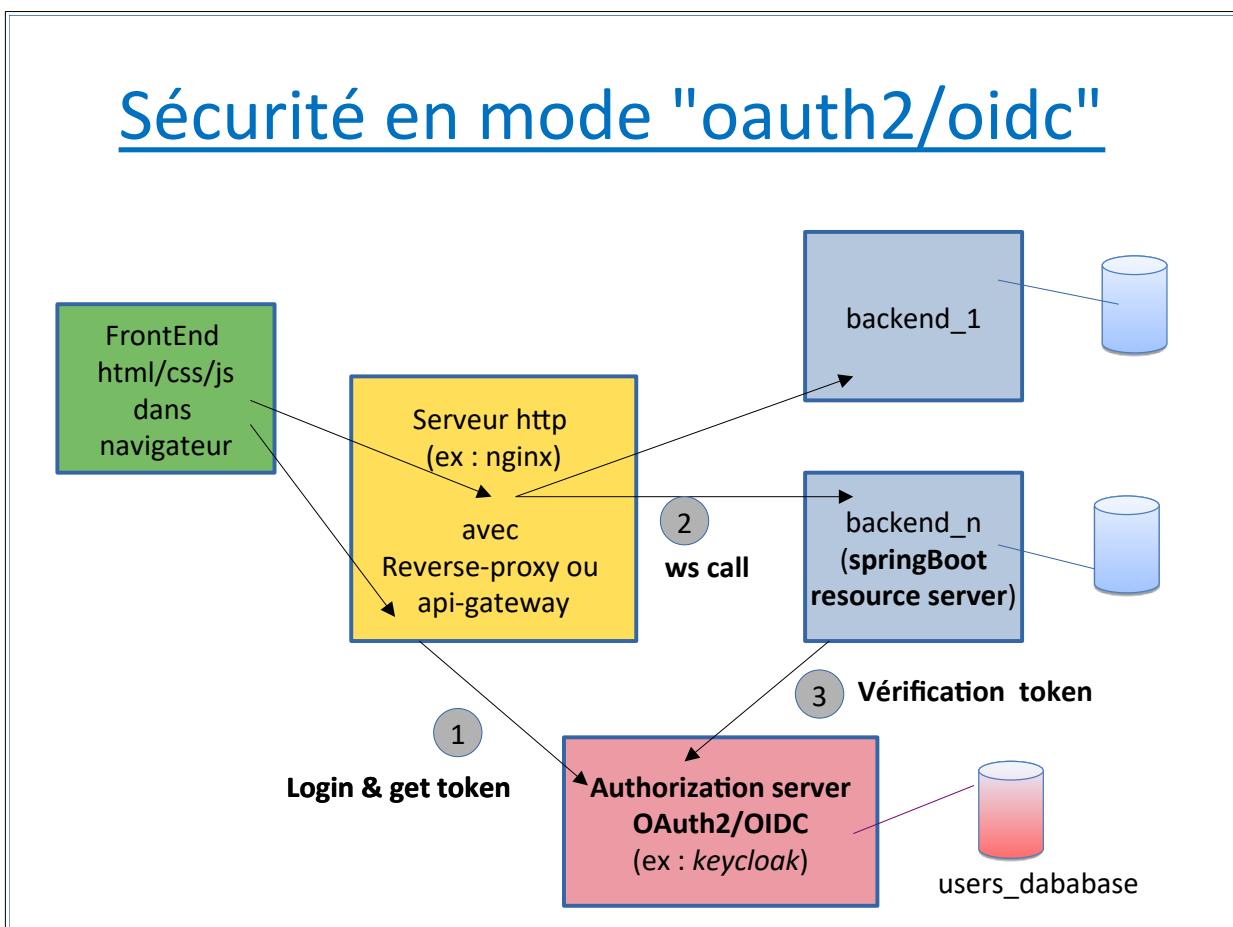
Sécurité en mode "jee/dhtml"



Sécurité en mode "standalone jwt"

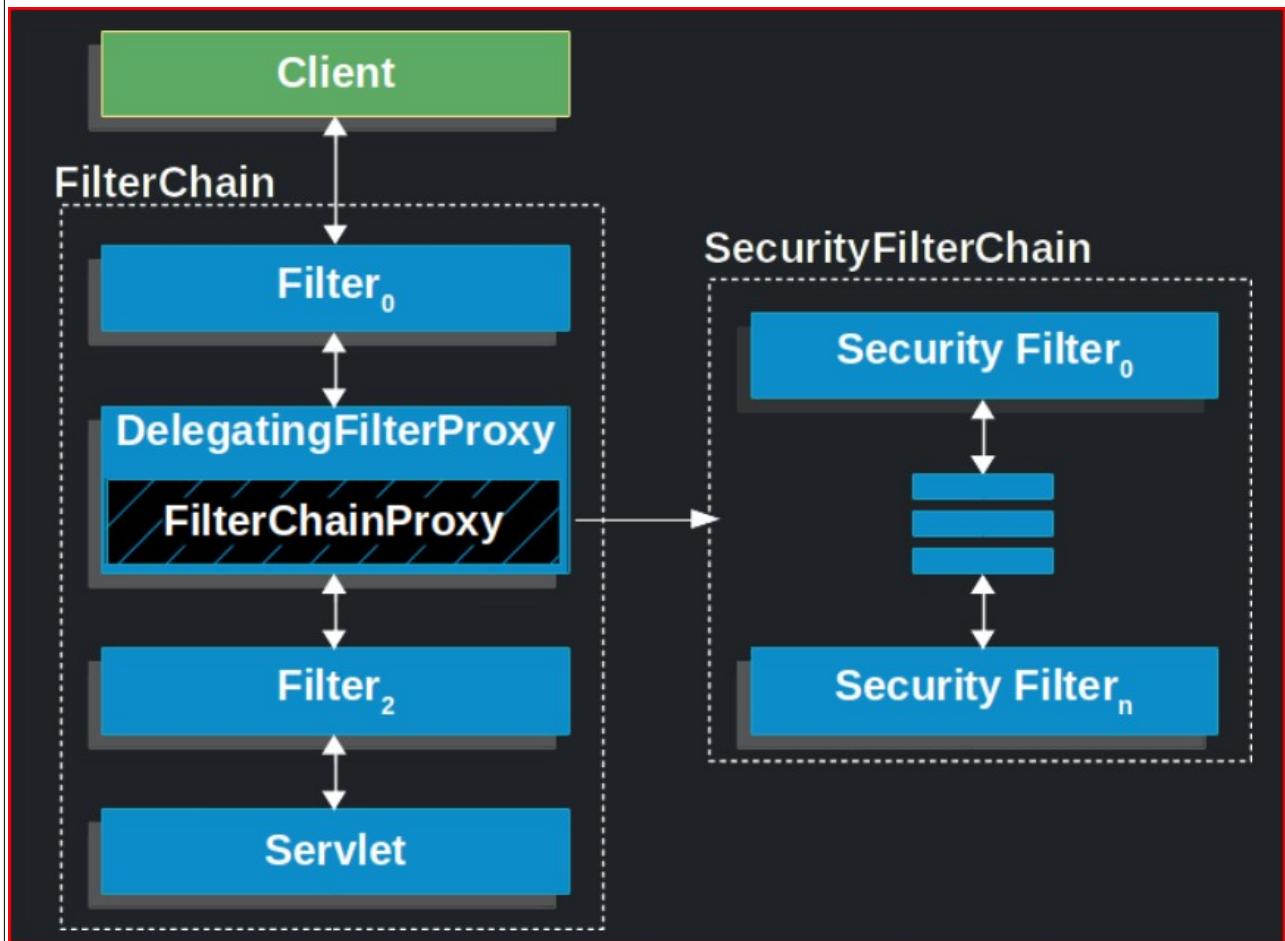


Sécurité en mode "oauth2/oidc"



1.3. Filtre web et SecurityFilterChain

SecurityFilterChain as Web Filter



En interne les principales technologies "web" de spring sont basées sur des Servlets (Http).

Exemple : SpringMvc avec DispatcherServlet .

Avant qu'une requête Http soit traitée par Spring-Mvc et le DispatcherServlet , on peut configurer des filtres web (respectant l'interface normalisée javax.servlet.Filter) qui vont intercepter cette requête de manière à effectuer des pré-traitements (et d'éventuels post-traitements).

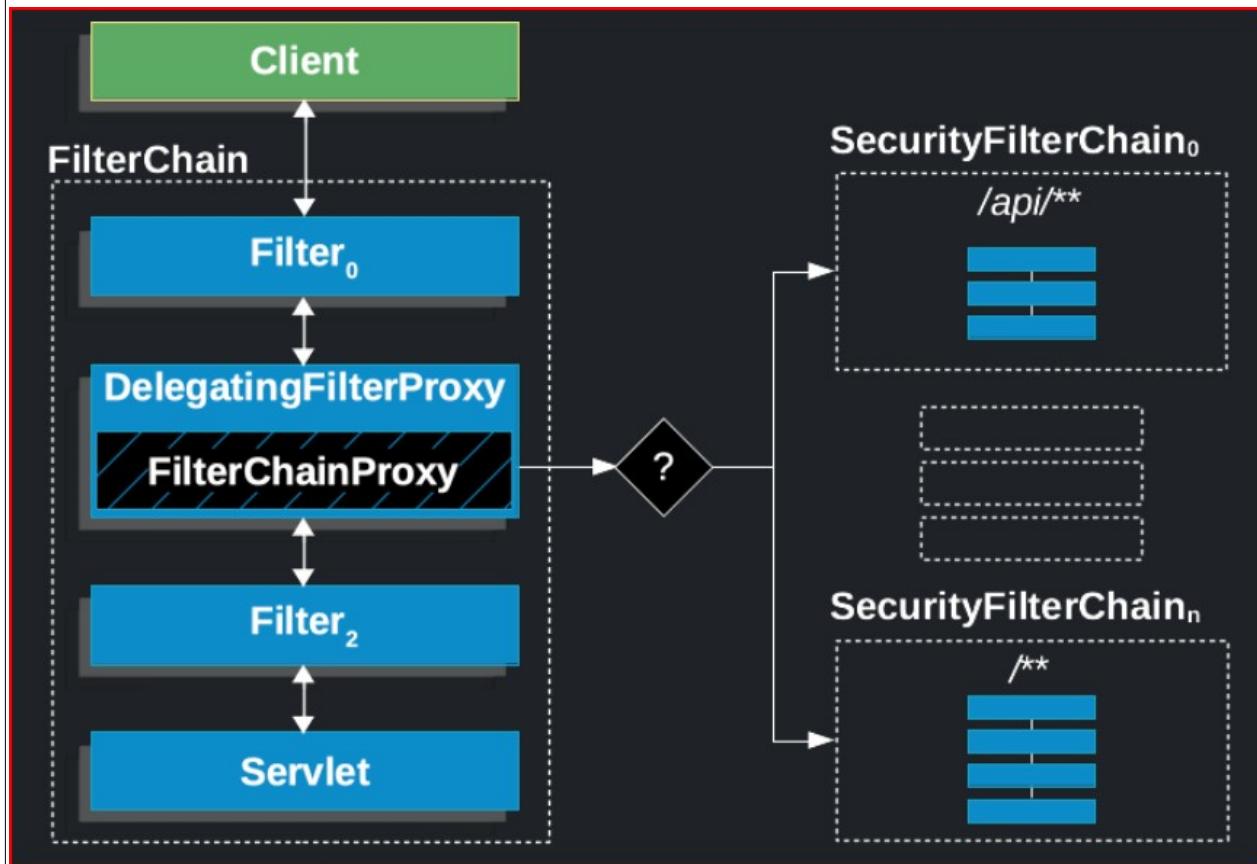
Au sein d'une application Spring ou SpringBoot, le composant prédéfini "DelegatingFilterProxy / FilterChainProxy" va automatiquement intercepter une requête Http et déclencher une chaîne ordonnée de "SecurityFilter" .

La "SecurityFilterChain" peut être unique dans le cas d'une application bien précise (ex1 : AppliWeb uniquement basée sur JSP/thymeleaf , ex2 : Api-rest en mode micro-service)

Une application Spring/SpringBoot peut cependant comporter plusieurs parties complémentaires et il est alors possible de configurer plusieurs "SecurityFilterChain" .

1.4. Multiple SecurityFilterChain

Multiple SecurityFilterChain (1/2)



NB: quand une requête arrive, le FilterChainProxy de Spring-security va utiliser le premier SecurityFilterChain correspondant à l'url de la requête et va ignorer les autres (point clef: la correspondance se fait via httpSecurity.antMatcher() sans s)

Il est donc important qu'une partie de l'URL (plutôt au début) puisse faire office d'aiguillage non ambigu vers une SecurityFilterChain ou une autre.

Exemple de convention d'URL :

`/rest/api-xyz/...`

ou

`/site/...`

ou

`**`

Multiple SecurityFilterChain (2/2)

```

@Configuration
public class MySecurityConfig {

    @Bean  @Order(1)
    protected SecurityFilterChain restApiFilterChain(
        HttpSecurity http) throws Exception {
        http.securityMatcher("/rest/**")
            .authorizeHttpRequests(...).build();
    }

    @Bean  @Order(2)
    protected SecurityFilterChain siteFilterChain(
        HttpSecurity http) throws Exception {
        http.securityMatcher("/site/**")
            .authorizeHttpRequests(...).build();
    }

    @Bean  @Order(3)
    protected SecurityFilterChain othersFilterChain(
        HttpSecurity http) throws Exception {
        http.securityMatcher("/**") // "/**" in last order !!!
            .authorizeHttpRequests(...).build();
    }
}

```

NB: 3 securityChain avec ordre important à respecter

- @Order(1) pour les URL commençant par /rest (ex: /rest/api-xxx , /rest/api-yyy)
- @Order(2) pour une éventuelle partie /site/ basée sur @Controller + JSP ou Thymeleaf
- @Order(3) ou @Order(99) pour le reste (autres URLs , pages static ou pas "spring")

NB : une instance de SecurityFilterChain peut éventuellement être associée à un "AuthenticationManager" spécifique ou bien ne pas l'être et dans le cas un AuthenticationManager global/principal sera utilisé par défaut .

1.5. Vue d'ensemble sur les phases de Spring-security

1. Une des premières phases exécutées par un filtre de sécurité consiste à **extraire certaines informations d'authentification de la requête Http** (ex : `username/password` en mode "basic" ou bien **jeton** (jwt ou autre) en mode "bearer").
2. Une seconde phase consiste à déclencher `authManager.authenticate(authentication_to_check)` de manière à comparer les informations d'authentification à vérifier avec une liste d'utilisateurs valide (à récupérer quelquepart : LDAP , JDBC , InMemory , OAuth2/OIDC, ...)
3. Les informations sur l'authentification réussie sont stockée dans un point central `SecurityContextHolder.getContext()` au format **Authentication** (interface avec variantes)
4. Certaines configurations "xml" ou "java" ou "via annotations" précises permettront d'accepter ou refuser un traitement demandé en fonction des informations d'authentification réussies stockées préalablement dans le contexte de sécurité.
(ex : `@PreAuthorize("hasRole('ADMIN')")`
ou bien `@PreAuthorize("hasAuthority('SCOPE_resource.delete')")`)

1.6. Comportement de l'authentification (spring-security)

L'interface fondamentale "**AuthenticationManager**" comporte la méthode fondamentale **authenticate()** dont le comportement est ci-après expliqué :

Authentication authenticate(Authentication authentication) throws AuthenticationException;

avant appel : authentication avec `getPrincipal()` retournant souvent `username (String)`

`getCredential()` retournant `password` à tester ou autre .

après appel : authentication avec `getPrincipal()` retournant `UserDetails` si ok

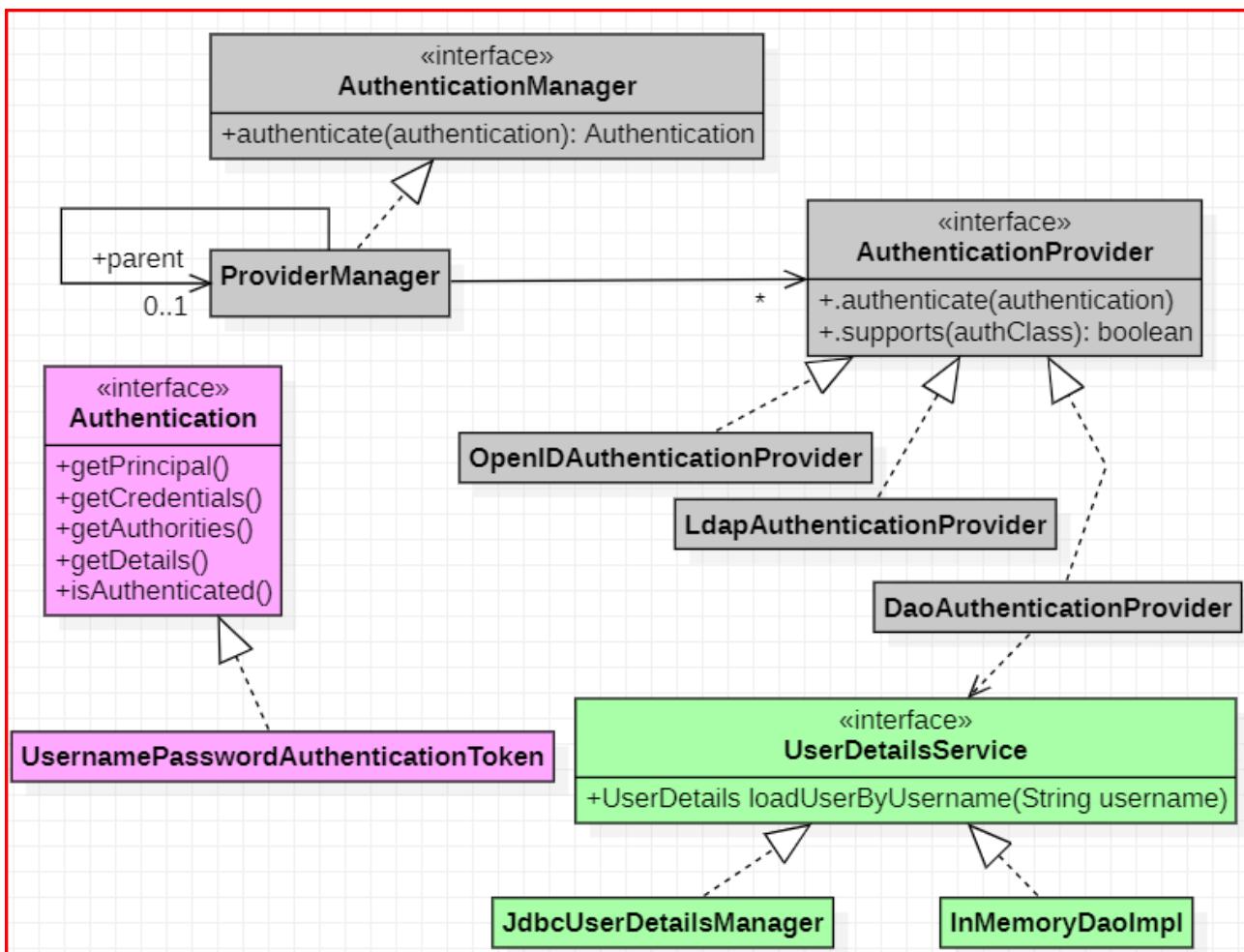
ou bien `AuthenticationException` sinon

Si l'authentification échoue --> `AuthenticationException` --> retour status HTTP 401
(Unauthorized) ou bien redirection vers formulaire de login (en fonction du contexte)

Si l'authentification est réussie -->

- la méthode `authenticate()` retourne un objet (implémentant l'interface "Authentication") bien complet (comportant "Roles utilisateurs" , ...).
- L'objet "Authentication" est alors automatiquement stocké dans le "SecurityContextHolder / SecurityContext" (lié au Thread courant prenant en charge la requête Http) par spring-security .

1.7. Mécanismes d'authentification (spring-security)



NB1: La classe "ProviderManager" implémente l'interface `AuthenticationManager` en itérant sur une liste de `AuthenticationProvider` enregistrés de façon à trouver le premier `AuthenticationProvider` capable de gérer l'authentification.

NB2 : A priori , Le "ProviderManager" principal (lié à l'implémentation de `AuthenticationManager`) est potentiellement relié à un `ProviderManager` parent (qui n'est utilisé que si l'authentification réalisée par le "AuthenticationManager/ ProviderManager" échoue).

Ce lien s'effectue via `AuthenticationManagerBuilder.parentAuthenticationManager()`

La classe `DaoAuthenticationProvider` correspond à une implémentation importante de `AuthenticationProvider` qui s'appuie en interne sur `UserDetailsService` (`Jdbc` ou `InMemory` ou spécifique)

1.8. Vue d'ensemble sur configuration concrète de la sécurité

Historique important :

- vers 2010 , configuration de spring-security au format xml (via un fichier spring-security.xml)
et balises de types <security-http> , <security:intercept-url , permitAll , denyAll , /> ,
<security:authentication-manager> <security:authentication-provider> <security:user-service> <security:user name="user1" password="pwd1" authorities="ROLE_USER" /> .

- Vers 2015-2020 , configuration souvent "java" de la sécurité via

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    .... protected void configure(final HttpSecurity http) throws Exception {...}
}
```

- Depuis 2022 et Spring 5.7 WebSecurityConfigurerAdapter est devenu "deprecated/obsolete" et il est conseillé d'utiliser

```
@Configuration
public class MySecurityConfig /* without inheritance */ {
    ... protected void SecurityFilterChain
        myFilterChain(HttpSecurity http) throws Exception {...}
}
```

- Depuis 2023 et Spring 6 , SpringSecurity a encore évolué :

Plus de `.and()`- mais que des lambda-expressions imbriquées plus claires (mieux délimitées) .

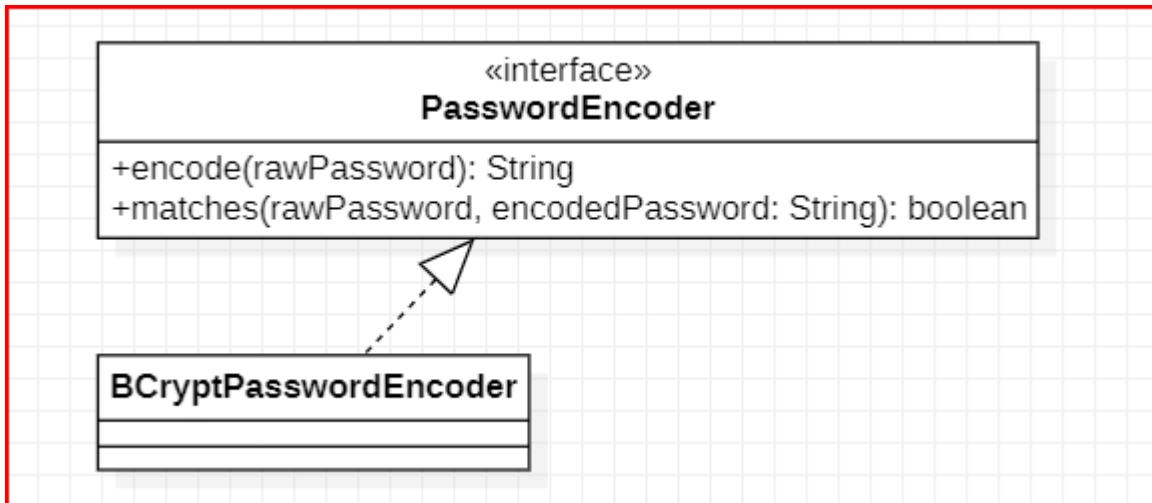
Quelques autres changements:

`antMatchers()` remplacé par `requestMatchers(...)`
`@EnableGlobalMethodSecurity(prePostEnabled = true)` remplacé apr
`@EnableMethodSecurity()` avec `prePostEnabled = true` par défaut
...
...

Dans tous les cas , `HttpSecurity http` , correspond à un point centralisé de la configuration de spring-security qu'il faut :

- soit analyser (si d'origine XML)
- soit définir et construire dans le cas d'une configuration "java" (@Configuration)

1.9. Encodage classique des mots de passe via BCrypt



NB : L'algorithme de cryptage "BCrypt" a été spécialement mis au moins pour encoder des mots de passes avant de les stocker en base. A partir d'un encodage "bcrypt" il est quasi impossible de déterminer le mot de passe d'origine qui a été crypté .

NB : Via BCrypt , si on encodage plusieurs fois "pwd1" , ça va donner des encodages différents (ex : "\$2a\$10\$wdysBwvK8l5t5zJsuKdcu.wMJJum8f3BA5/X6muaNpVoLx4rj1tKm" ou "\$2a\$10\$OBiZKdISPSio6LI7Mh9eRubBVIQ8q0NzCoSIcDIm9L4MvBzwmbfmq")

Ceci dit via la méthode **.matches()** les 2 encodages seront tous les 2 considérés comme corrects pour tester le mot de passe "pwd1" .

```

.....
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
@Configuration
public class MySecurity {
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
        //or new BCryptPasswordEncoder(int strength) with strength between 4 and 31
    }
}
  
```

1.10. Prise en compte d'une authentification vérifiée

Une fois l'authentification effectuée et stockée dans le contexte "SecurityContextHolder", on peut alors très facilement accéder aux infos "utilisateur" vérifiées via des instructions de ce type :

```
Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();
if (principal instanceof UserDetails) {
    String username = ((UserDetails)principal).getUsername();
}
```

L'objet "Authentication" comporte une méthodes **getAuthorities()** retournant un paquet d'éléments de type "GrantedAuthority" dont "SimpleGrantedAuthority" est l'implémentation la plus classique. "SimpleGrantedAuthority" comporte un nom de rôle (ex "ROLE_ADMIN" ou "ROLE_USER" , ...)

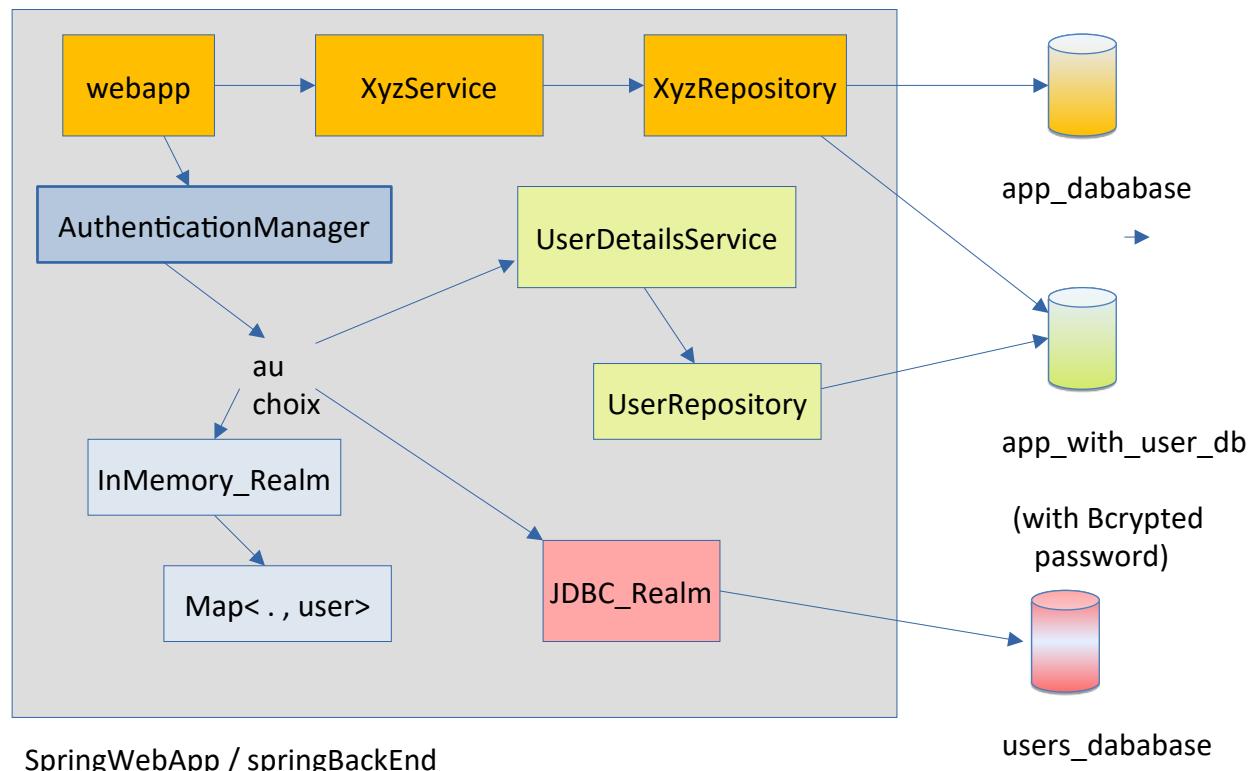
Lorsqu'un peu plus tard , un accès à une partie de l'application sera tenté (page jsp , méthode appelée sur un contrôleur , ...) les mécanismes de la partie "contrôle d'accès" de spring-security pour alors assez facilement autoriser ou refuser les actions en comparant les rôles mémorisés dans l'objet "Authentication" du contexte avec certaines configurations du genre :

```
@PreAuthorize("hasRole('ADMIN')")
```

2. Configuration des "Realms" (spring-security)

2.1. Principales implémentations possibles des realms

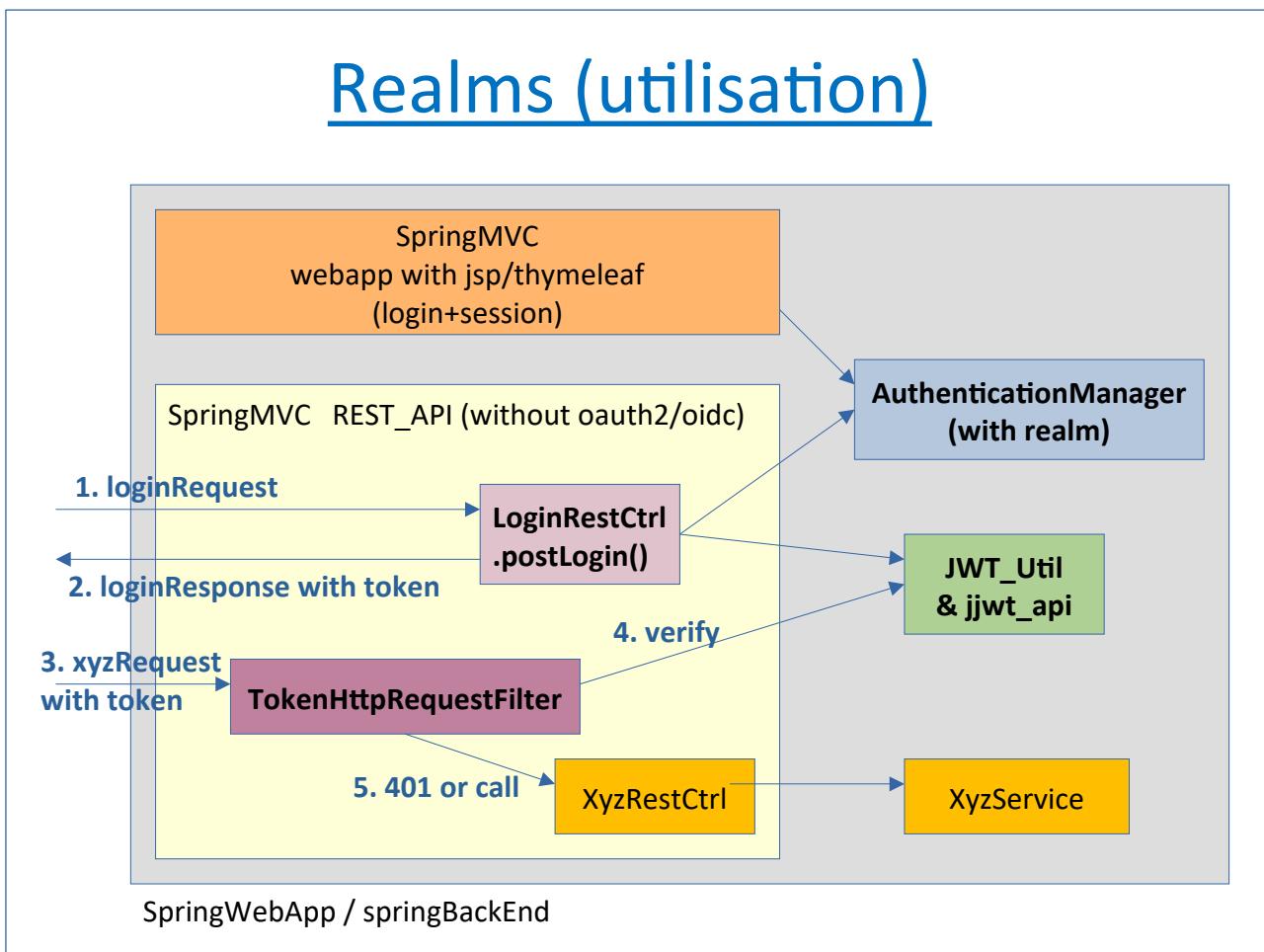
Realms (implémentations possibles)



JDBC_Role	Pour accès jdbc/sql à des tables users et roles dans une base de données dédiées (utilisateurs)
InMemory_Role	Pour tests rapides/simples en phase de développement
UserDetailsService	Pour accès personnalisé aux utilisateurs (via code habituel JPA au autre) → implémentation libre/souple (cas le plus fréquent)
Autres (LDAP, ...)	...

Ces implémentations sont presque toujours exclusives (bien qu'une combinaison soit possible).

2.2. Utilisation classique d'un realm



Sans serveur OAuh2/OIDC , le backend spring doit lui-même :

- vérifier le login via l'authenticationManager (avec souvent en arrière plan des "bcrypted_password")
- construire un jeton (souvent au format JWT)
- retourner ce jeton au sein de loginResponse
- extraire un jeton d'une requête api et vérifier si celui-ci est valide
- retourner 401/Unauthorized si le jeton est invalide (ou si pas de jeton)
- extraire du jeton JWT les infos sur l'utilisateur authentifié (username , rôle , ...)
- invoquer le WS_REST fonctionnel (Xyz) si le jeton est valide
- vérifier le rôle requis via `@PreAuthorize(hasRole(...))` et retourner 403/Forbidden si problème
- construire et retourner la réponse HTTP/JSON si tout est ok

2.3. AuthenticationManagerBuilder

L'objet technique ***AuthenticationManagerBuilder*** sert à construire un objet implémentant l'interface ***AuthenticationManager*** qui servira lui même à authentifier l'utilisateur .

Selon le contexte de l'application, cet objet fondamental peut être récupéré de l'une des façons suivantes :

- par injection de dépendances (si déjà préparé/défini ailleurs)
- **par instantiation directe**
- par récupération dans la partie "*sharedObject*" de *HttpSecurity*

Exemples (à adapter au contexte) :

```
public static AuthenticationManagerBuilder newAuthenticationManagerBuilder() {
    final ObjectPostProcessor<Object> objectPostProcessor =
        new ObjectPostProcessor<Object>() {
            @Override
            public <O extends Object> O postProcess(final O object) {
                return object;
            }
        };
    AuthenticationManagerBuilder authMgrBuilder =
        new AuthenticationManagerBuilder(objectPostProcessor);
    return authMgrBuilder;
}
```

```
public static AuthenticationManagerBuilder authenticationManagerBuilderFromHttpSecurity(HttpSecurity httpSecurity) {
    AuthenticationManagerBuilder authenticationManagerBuilder = httpSecurity
        .getSharedObject(AuthenticationManagerBuilder.class);
    authenticationManagerBuilder.parentAuthenticationManager(null);
    return authenticationManagerBuilder;
}
```

NB : Une fois créé ou récupéré , cet objet "AuthenticationManagerBuilder" sera la base souvent indispensable du paramétrage d'un "realm" (liste d'utilisateurs autorisés à utiliser l'application).

2.4. Délégation d'authentification (OAuth2/Oidc)

```
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
...
public HttpSecurity configureEndOfSecurityChain(HttpSecurity http) throws Exception{
    return http
        .sessionManagement(sM →
            sM.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .oauth2ResourceServer((oauth2) → oauth2.jwt(Customizer.withDefaults()));
}
```

et

dans application.properties

```
spring.security.oauth2.resource-server.jwt.issuer-uri=https://www.d-defrance.fr/keycloak/realmns/sandboxrealm
```

et

```
@PreAuthorize("hasAuthority('SCOPE_resource.write')") ou autre
```

avec dans pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

2.5. Realm temporaire "InMemory"

```
authenticationManagerBuilder.inMemoryAuthentication()
    .withUser("user1").password(passwordEncoder.encode("pwd1")).roles("USER").and()
    .withUser("admin1").password(passwordEncoder.encode("pwd1")).roles("ADMIN").and()
    .withUser("user2").password(passwordEncoder.encode("pwd2")).roles("USER").and()
    .withUser("admin2").password(passwordEncoder.encode("pwd2")).roles("ADMIN");
```

2.6. Authentification jdbc ("realm" en base de données)

La configuration ci-après permet de configurer **spring-security** pour qu'il accède à une **liste de comptes "utilisateurs" dans une base de données relationnelle** (ex : H2 ou Mysql ou ...).

Cette base de données sera éventuellement différente de celle utilisée par l'aspect fonctionnel de l'application .

Au sein de l'exemple suivant , la méthode **initRealmDataSource()** paramètre un objet DataSource vers une base h2 spécifique à l'authentification (**jdbc:h2:~/realmdb**).

L'instruction

```
JdbcUserDetailsManagerConfigurer jdbcUserDetailsManagerConfigurer =
    auth.jdbcAuthentication().dataSource(realmDataSource);
```

permet d'initialiser AuthenticationManagerBuilder en mode jdbc en précisant le DataSource et donc la base de données à utiliser .

L'instruction jdbcUserDetailsManagerConfigurer.**withDefaultSchema()**; (*à ne lancer que si les tables "users" et "authorities" n'existent pas encore dans la base de données*) permet de créer les tables nécessaires (*avec noms et structures par défaut*) dans la base de données.

Par défaut , la table **users(username, password)** comporte les mots de passe (souvent cryptés) et la table **authorities(username, authority)** comporte la liste des rôles de chaque utilisateur

JdbcAppDbGlobalUserDetailsConfig.java à adapter au contexte

```
package org.mycontrib.generic.security.config;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.ResultSet;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.authentication.configurers.provisioning.JdbcUserDetailsManagerConfigurer;
import org.springframework.security.config.annotation.authentication.configurers.provisioning.UserDetailsManagerConfigurer;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
//@Profile("appDbSecurity") //with jdbc
public class JdbcAppDbGlobalUserDetailsConfig {
    @Autowired
    private BCryptPasswordEncoder passwordEncoder;

    private static DataSource realmDataSource;

    private static void initRealmDataSource() {
        DriverManagerDataSource driverManagerDataSource = new DriverManagerDataSource();
        driverManagerDataSource.setDriverClassName("org.h2.Driver");
        driverManagerDataSource.setUrl("jdbc:h2:~/realmdb");
        driverManagerDataSource.setUsername("sa");
        driverManagerDataSource.setPassword("");
        realmDataSource = driverManagerDataSource;
    }

    private boolean isRealmSchemaInitialized() {
        int nbExistingTablesOfRealmSchema = 0;
        try {
            Connection cn = realmDataSource.getConnection();
            DatabaseMetaData meta = cn.getMetaData();
            String tabOfType[] = {"TABLE"};
            ResultSet rs = meta.getTables(null,null,"%",tabOfType);
            while(rs.next()){


```

```

        String existingTableName = rs.getString(3);
        if(existingTableName.equalsIgnoreCase("users")
           || existingTableName.equalsIgnoreCase("authorities")) {
            nbExistingTablesOfRealmSchema++;
        }
    }
    rs.close();
    cn.close();
} catch (Exception e) {
    e.printStackTrace();
}
return (nbExistingTablesOfRealmSchema>=2);
}

@Autowired
public void globalUserDetails(final AuthenticationManagerBuilder auth) throws Exception {
    initRealmDataSource();
    JdbcUserDetailsManagerConfigurer jdbcUserDetailsManagerConfigurer =
        auth.jdbcAuthentication().dataSource(realmDataSource);
    if(isRealmSchemaInitialized()) {
        /*
         * jdbcUserDetailsManagerConfigurer
         * .usersByUsernameQuery("select username,password, enabled from users where username=?")
         * .authoritiesByUsernameQuery("select username, authority from authorities where username=?");
         * //by default
         */
        // or .authoritiesByUsernameQuery("select username, role from user_roles where username=?")
        //if custom schema
    } else {
        //creating default schema and default tables "users" , "authorities"
        jdbcUserDetailsManagerConfigurer.withDefaultSchema();
        //insert default users:
        configureDefaultUsers(jdbcUserDetailsManagerConfigurer);
    }
}

void configureDefaultUsers(UserDetailsManagerConfigurer udmc){
    udmc
        .withUser("user1").password(passwordEncoder.encode("pwduser1")).roles("USER").and()
        .withUser("admin1").password(passwordEncoder.encode("pwdadmin1")).roles("ADMIN","USER").and()
        .withUser("publisher1").password(passwordEncoder.encode("pwdpublisher1")).roles("PUBLISHER","USER").and()
        .withUser("user2").password(passwordEncoder.encode("pwduser2")).roles("USER").and()
        .withUser("admin2").password(passwordEncoder.encode("pwdadmin2")).roles("ADMIN").and()
        .withUser("publisher2").password(passwordEncoder.encode("pwdpublisher2")).roles("PUBLISHER");
}
}

```

2.7. Authentification "personnalisée" en implémentant l'interface UserDetailsService

Si l'on souhaite coder un accès spécifique à la liste des comptes utilisateurs (ex : via JPA ou autres), on peut implémenter l'interface **UserDetailsService** .

L'interface **UserDetailsService** comporte cette unique méthode :

```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
```

Cette méthode est censée remonter les données d'un compte utilisateur depuis un certain endroit (base de données , mongoDB ,) .

Ces infos "utilisateur" doivent être une implémentation de l'interface "**UserDetails**" (classe "User" par exemple). L'objet "User" (ou un équivalent implémentant "UserDetails") est censée comporter le bon mot de passe.

Les mécanismes internes de Spring-security ("AuthenticationProvider" , ...) vont alors pouvoir comparer le bon mot de passe avec celui renseigné par l'utilisateur qui souhaite s'authentifier.

Dans certains cas la comparaison passe par une implémentation de "PasswordEncoder" (ex : "BCryptPasswordEncoder") lorsque les mots de passe sont cryptés dans la base de données.

Exemple :

```
package ....;

import ...;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.password.PasswordEncoder;

@Profile("withSecurity")
@Service
public class MyUserDetailsService implements UserDetailsService {
    Logger logger = LoggerFactory.getLogger(MyUserDetailsService.class);
    @Autowired private PasswordEncoder passwordEncoder;
    @Autowired private ServiceCustomer serviceCustomer;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        UserDetails userDetails=null;
        logger.debug("MyUserDetailsService.loadUserByUsername() called with username="+username);

        List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
        String password=null;
        if(username.equals("james_Bond")) {
            password=passwordEncoder.encode("007");//simulation password ici
            authorities.add(new SimpleGrantedAuthority("ROLE_AGENTSECRET"));
        }
    }
}
```

```

        userDetails = new User(username, password, authorities);
    }
    else {
        //NB le username considéré comme potentiellement
        //égal à firstname_lastname
        try {
            String firstname = username.split("_")[0];
            String lastname = username.split("_")[1];

            List<Customer> customers =
                serviceCustomer.rechercherCustomerSelonPrenomEtNom(firstname,lastname);
            if(!customers.isEmpty()) {
                Customer firstCustomer = customers.get(0);
                authorities.add(new SimpleGrantedAuthority("ROLE_CUSTOMER"));
                //ou "ROLE_USER" ou "ROLE_ADMIN"
                password=firstCustomer.getPassword(); // déjà stocké en base en mode crypté
                //password=passwordEncoder.encode(firstCustomer.getPassword());
                //si pas stocké en base en mode crypté (PAS BIEN !!!)
                userDetails = new User(username, password, authorities);
            }
            } catch (Exception e) {
                //e.printStackTrace();
            }
        }

        if(userDetails==null) {
            //NB: il est important de remonter UsernameNotFoundException (mais pas null , ni une autre exception)
            //si l'on souhaite qu'en cas d'échec avec cet AuthenticationManager
            //un éventuel AuthenticationManager parent soit utilisé en plan B
            throw new UsernameNotFoundException(username + " not found");
        }

        return userDetails;
    }
}

```

3. Configuration des zones(url) à protéger

3.1. Généralités sur la configuration de HttpSecurity

Ancienne façon de faire (devenue obsolète/deprecated depuis la version 5.7 de spring-security) :

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    ...
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/", "/favicon.ico", "/**/*.png", "/**/*.gif", "/**/*.svg",
                "/**/*.jpg", "/**/*.css", "/**/*.map", "/**/*.js").permitAll()
            .antMatchers("/to-welcome").permitAll()
            .antMatchers("/session-end").permitAll()
            .antMatchers("/xyz").permitAll()
            .anyRequest().authenticated()
            .and().formLogin().permitAll()
            .and().csrf();
    }
    ...
}
```

Nouvelle façon conseillée depuis la version 5.7 de spring-security (ici en syntaxe v6) :

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WithSecurityMainFilterChainConfig {
    ...
    @Bean
    @Order(99)
    protected SecurityFilterChain myFilterChain(HttpSecurity http)
        throws Exception {
        http.authorizeHttpRequests(
            //exemple très permissif ici à grandement adapter !!!!
            auth -> auth.requestMatchers("/**/*.*").permitAll() )
            .cors(Customizer.withDefaults())
            .headers(headers ->
                headers.frameOptions( frameOptions-> frameOptions.sameOrigin() ) )
            .csrf(csrf->csrf().disable());
        return http.build();
    }
}
```

NB : la méthode `myFilterChain()` pourra éventuellement appeler des sous fonctions pour paramétrier http (de type HttpSecurity) de façon flexible et modulaire avant de déclencher `http.build()`.

3.2. Configuration type pour un projet de type Thymeleaf ou JSP

```
package ....;
...
{
    public HttpSecurity configureHttpSecurityV1(HttpSecurity http) throws Exception {

        return http.authorizeHttpRequests( auth ->
            auth.requestMatchers("/", "/favicon.ico",
                "/**/*.png",
                "/**/*.gif",
                "/**/*.svg",
                "/**/*.jpg",
                "/**/*.css",
                "/**/*.map",
                "/**/*.js").permitAll()
            .requestMatchers("/to-welcome").permitAll()
            .requestMatchers("/session-end").permitAll()
            .requestMatchers("/xyz").permitAll()
            .anyRequest().authenticated() )
        .formLogin( formLogin -> formLogin.permitAll() )
        /*formLogin( formLogin -> formLogin.loginPage("/login")
                    .failureUrl("/login-error")
                    .permitAll() */

        .csrf(Customizer.withDefaults());
    }
}
```

3.3. Champ caché "_csrf" de spring-mvc utile pour pages/vues "java/jsp" mais inutile pour Api-REST avec tokens .

NB: Ce champ caché correspond au "*Synchronizer Token Pattern*" (que l'on retrouve dans les frameworks web concurrents "Stuts" ou "JSF") : le coté serveur compare la valeur d'un jeton aléatoire stockée en session http avec celle stockée dans un champ caché et refuse de gérer la requête "re-postée" si la comparaison n'est pas réussie.

D'autre part , le terme **CSRF** (signifiant "*Cross Site Request Forgery*" correspond à un éventuel problème de sécurité : un site "malveillant" (utilisé en parallèle au sein d'un navigateur) déclenche automatiquement (via javascript ou autre) des requêtes non voulues (ex : virement monétaire) en utilisant le contexte d'un site à priori de confiance (mais pas assez protégé) .

Avec <form> (au lieu de <form:form> de SpringMvc / jsp) , il faut insérer nous même le champ suivant au sein du formulaire d'une page ".jsp" :

```
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
```

<form:form ...> de SpringMvc / jsp ou bien l'équivalent thymeleaf gère (génère) automatiquement le champ caché _csrf attendu par **spring-security** . Exemple : <input type="hidden" name="_csrf" value="8df91b84-74c1-4013-bd44-ede7b00779a2" />) .

3.4. Configuration type pour un projet de type "Api REST"

```

package ....;
...
{
    public HttpSecurity configureHttpSecurityV2(HttpSecurity http) throws Exception {
        return http.authorizeHttpRequests( auth ->
            auth.requestMatchers("/", "/favicon.ico", "/**/*.png", "/**/*.gif", "/**/*.svg",
                "/**/*.jpg", "/**/*.html", "/**/*.css", "/**/*.js").permitAll()
            .requestMatchers(HttpServletRequestMethod.POST,"/auth/**").permitAll()
            .requestMatchers("/xyz-api/public/**").permitAll()
            .requestMatchers("/xyz-api/private/**").authenticated() )
        .cors( Customizer.withDefaults())
        //enable CORS (avec @CrossOrigin sur class @RestController)
        .csrf( csrf -> csrf.disable() )
        // If the user is not authenticated, returns 401
        .exceptionHandling(eh ->
            eh.authenticationEntryPoint(getRestAuthenticationEntryPoint() ) )
        // This is a stateless application, disable sessions
        .sessionManagement(sM ->
            sM.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        // Custom filter for authenticating users using tokens
        .addFilterBefore(jwtAuthenticationFilter,
            UsernamePasswordAuthenticationFilter.class);
    }

    private AuthenticationEntryPoint getRestAuthenticationEntryPoint() {
        return new HttpStatusEntryPoint(HttpStatus.UNAUTHORIZED);
    }
}
...

```

Spécificité importante de Spring6 :

application.properties

```

...
#pour si besoin interptreter des ** dans SecurityConfig en spring 6 exactement comme en spring 5
spring.mvc.pathmatch.matching-strategy=ANT_PATH_MATCHER
#spring.mvc.pathmatch.matching-strategy=ant-path-matcher
#spring.mvc.pathmatch.matching-strategy=path-pattern-parser

```

path-patter-parser (utilisé maintenant par SpringBoot 3) est plus performant (en analysant les "path" dès le démarrage de l'application) mais "xxx/**" doit être reformulé en "xxx" plus "xxx/**".

3.5. Paramétrage des autorisations selon rôles ou scopes

```
import org.springframework.security.access.prepost.PreAuthorize;
...
@DeleteMapping("/{id}")
@PreAuthorize("hasAuthority('SCOPE_resource.delete')")
//ou bien @PreAuthorize("hasRole('ADMIN')")
public ResponseEntity<?> deleteXyzById(@PathVariable("id") Long id) {
...
}
```

Rôle	Rôle d'un sujet authentifié (classique en mode JEE , quelquefois possible avec OIDC selon configuration)
Scope	Un des droits élémentaires associé à un sujet authentifié en mode OAuth2/OIDC

→ Si l'utilisateur n'a pas le rôle (ou le scope) requis → **403 / Forbidden** .

3.6. Eventuelle configuration d'autorisations "CORS"

```
import org.springframework.web.bind.annotation.CrossOrigin ;
...
@RestController
@RequestMapping(value="/rest/api-xyz/v1/xyz" , headers="Accept=application/json")
@CrossOrigin(origins = "*", methods = { RequestMethod.GET , RequestMethod.POST ,
RequestMethod.PATCH , RequestMethod.DELETE , RequestMethod.PUT})
public class XyzRestCtrl { ... }
```

ou plus finement **origins = { "www.aaa.com" , "www.bbb.com" }**

Cas où @CrossOrigin peut être utile	- début de dev du frontEnd (appels directs de WS REST en HTTP/ajax) - appels ajax directs depuis d'autres entreprises (autres noms de domaine)
Cas où @CrossOrigin est <u>inutile</u>	- appels ajax indirects via "reverseProxy" ou "api-gateway" (souvent le cas en production)

XII - Tests avancés avec Spring

1. Différents types de tests (environnement spring)

- **Test purement unitaire** (*à portée très limitée*) : on ne teste qu'une toute petite partie de l'application spring (ex : quelques méthodes d'un service) en simulant (via des "mocks" le ou les DAOs en arrière plan)
- **Test d'intégration de type "end-to-end"** : on teste l'ensemble de l'application spring (c'est à dire toute la chaîne "Api-REST + services_métiers + DAO + base_H2_ouAutre")
- **Test d'intégration partiel** : on teste une sous partie de l'application spring (exemples : "DAO + baseH2" ou bien "service_métier + DAO + baseH2" ou bien "Api-REST + mockDeServicesMetiers")
- ...

2. Test purement unitaire

Code à tester :

```
...
@Service
@Transactional
public class ServiceDeviseV2 implements ServiceDevise{
    private static Logger logger = LoggerFactory.getLogger(ServiceDeviseV2.class);

    private RepositoryDevise repositoryDevise;

    //injection de dépendance par constructeur
    public ServiceDeviseV2(RepositoryDevise repositoryDevise) {
        this.repositoryDevise=repositoryDevise;
        logger.debug("ServiceDeviseV2 instance="+this.toString()
            + " using repositoryDevise="+repositoryDevise.getClass().getName());
    }

    @Override
    public double convertir(double montant, String codeDeviseSource, String codeDeviseCible)
        throws NotFoundException {
        try {
            Devise deviseSource = repositoryDevise.findById(codeDeviseSource).get();
            Devise deviseCible = repositoryDevise.findById(codeDeviseCible).get();
            return montant * deviseCible.getChange() / deviseSource.getChange();
        } catch (Exception e) {
            e.printStackTrace();
            throw new NotFoundException("devise_not_found",e); //ameliorable en précision
        }
    }
    ...
}
```

TestAlgorithmiePurementUnitaire.java

```

package com.mycompany.xyz.test;

import static org.mockito.ArgumentMatchers.anyString;
import java.util.Optional;
import org.junit.jupiter.api.Assertions; import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test; import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks; import org.mockito.Mockito;
import org.mockito.Mock; import org.mockito.MockitoAnnotations;
import org.mockito.junit.jupiter.MockitoExtension;
import org.slf4j.Logger; import org.slf4j.LoggerFactory;

import com.mycompany.xyz.entity.Devise;
import com.mycompany.xyz.repository.RepositoryDevise;
import com.mycompany.xyz.service.ServiceDeviseV2;

@ExtendWith(MockitoExtension.class) //for JUnit 5
public class TestAlgorithmiePurementUnitaire {
    //QUOI, des Mocks presque partout, mais de qui se moque-t-on ?

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

    @InjectMocks
    /* @InjectMocks pour demander à "Mockito" (sans spring) de :
       - créer une instance normale de cette classe (ici new ServiceDeviseV2(...))
       - d'injecter le ou les @Mock(s) de cette classe de test dans la classe ServiceDeviseV2()
         via un constructeur adéquat
    */
    private ServiceDeviseV2 serviceDevise;
    //à partiellement tester d'un point de vue purement algorithmique
    //et sans s'appuyer sur le contexte spring

    @Mock /* @Mock pour demander à "Mockito" (sans spring) de :
    - créer ultérieurement un Mock de l'interface
    - NB: il faudra appeler MockitoAnnotations.openMocks(this);
          pour initialiser tous les mocks de this préfixés par @Mock
    */

    private RepositoryDevise daoDeviseMock; //mock à utiliser

    @BeforeEach
    public void reInitMock() {
        //Mockito.initMocks(this); in old Junit 4
        MockitoAnnotations.openMocks(this); //with JUnit5/Jupiter
        /* MockitoAnnotations.openMocks(this) permet de créer des instances de chaque mock
           préfixé par @Mock au sein de this .
           ce qui revient au même que d'écrire :
           this.daoDeviseMock = Mockito.mock(RepositoryDevise.class);
           this.mock2=Mockito.mock(Interface_ouClasse2.class);
           s'il n'y avait pas d'utilisation de @Mock
        */
    }
}

```

```

@Test
public void testConvertir() {
    double montant=100;
    String codeDeviseSource="EUR";
    String codeDeviseCible="USD";
    double montantConverti=-1;
    //1.préparation du mock en arrière plan:
    Mockito.when(daoDeviseMock.findById(codeDeviseSource))
        .thenReturn(Optional.of(new Devise("EUR","Euro",1.0)));
    Mockito.when(daoDeviseMock.findById(codeDeviseCible))
        .thenReturn(Optional.of(new Devise("USD","Dollar",1.1)));
    //2.appel de la méthode convertir sur le service et test retour
    montantConverti = serviceDevise.convertir(montant, codeDeviseSource,
                                                codeDeviseCible);
    logger.debug("montantConverti="+montantConverti);
    Assertions.assertEquals(montant * 1.1 , montantConverti, 0.000001);
    //3.verif service appelant 2 fois deviseDao.findById() via aspect spy de Mockito:
    Mockito.verify(daoDeviseMock, Mockito.times(2)).findById(anyString());
}

@Test
public void testConvertirAvecDeviseInconnue() {
    double montant=100;
    String codeDeviseSource="EUR";
    String codeDeviseCibleInconnu="C??";
    double montantConverti=-1;
    //1.préparation du mock en arrière plan:
    Mockito.when(daoDeviseMock.findById(codeDeviseSource))
        .thenReturn(Optional.of(new Devise("EUR","Euro",1.0)));
    Mockito.when(daoDeviseMock.findById(codeDeviseCibleInconnu))
        .thenReturn(Optional.empty());
    //2.appel de la méthode convertir sur le service et test exception en retour
    try {
        montantConverti = serviceDevise.convertir(montant, codeDeviseSource,
                                                codeDeviseCibleInconnu);
        Assertions.fail("une exception aurait normalement du remonter");
    }catch(Exception ex) {
        logger.debug("exception normalement attendue="+ex.getMessage());
        Assertions.assertTrue(ex.getClass().getSimpleName()
                            .equals("NotFoundException"));
    }
}

```

3. Test spring avec DirtyContext

//Basic spring component to experiment @DirtiesContext() in test class

```
package com.mycompany.xyz.ex;
import java.util.HashSet;      import java.util.Set;
import org.slf4j.Logger;        import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

@Component
public class MySpringSetEx {
    private Set<String> exDataSet = new HashSet<>();

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

    public MySpringSetEx() {
        super();
        logger.debug("MySpringSetEx instance='"+this.toString());
    }

    public void addDataInExDataSet(String value) {
        exDataSet.add(value);
    }

    public Set<String> getExDataSet() {
        return this.exDataSet;
    }
}
```

TestWithOrWithoutDirtyContext.java

```
package com.mycompany.xyz.test;
import java.util.Set;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Order;      import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.slf4j.Logger;              import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.annotation.DirtiesContext;
import org.springframework.test.annotation.DirtiesContext.ClassMode;
import org.springframework.test.annotation.DirtiesContext.MethodMode;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import com.mycompany.xyz.MySpringBootApplication;
import com.mycompany.xyz.ex.MySpringSetEx;

@ExtendWith(SpringExtension.class)
@SpringBootTest(classes= {MySpringBootApplication.class})
@ActiveProfiles("embeddedDb")
public class TestWithOrWithoutDirtyContext {

    private static Logger logger = LoggerFactory.getLogger(TestWithOrWithoutDirtyContext.class);
```

@Autowired

```
private MySpringSetEx mySpringSetEx; //à tester/utiliser
```

@Test**@Order(1)**

```
public void addAndGetData1InMySpringContext() {
    mySpringSetEx.addDataInExDataSet("data_1a");
    mySpringSetEx.addDataInExDataSet("data_1b");
    Set<String> dataSet1 = mySpringSetEx.getExDataSet();
    logger.debug("dataSet1 = " + dataSet1); //dataSet1 = [data_1a, data_1b]
    Assertions.assertTrue(dataSet1.size() == 2);
}

/*
@Test
@Order(2)
public void addAndGetData2InMySpringContextWithoutDirtiesContextBadTest() {
    mySpringSetEx.addDataInExDataSet("data_2a");
    mySpringSetEx.addDataInExDataSet("data_2b");
    Set<String> dataSet2 = mySpringSetEx.getExDataSet();
    logger.debug("dataSet2 = " + dataSet2); //dataSet2 = [data_1a, data_2b, data_2a, data_1b]
    Assertions.assertTrue(dataSet2.size() == 2); //failing test: 4 != 2
}
*/
```

/*

@DirtiesContext demande à réinitialiser le contextSpring (et tout son contenu : tous ses composants) de façon à obtenir des tests aux comportements plus "unitaires"
 (sans effets de bord engendrés par les tests précédents)

Ne pas en abuser car cela peut ralentir l'exécution d'une séquence de tests

Si placé sur une méthode : `@DirtiesContext(methodMode = MethodMode.BEFORE_METHOD or MethodMode.AFTER_METHOD)`

Si placé sur la classe de test : `@DirtiesContext(classMode = ClassMode.BEFORE_CLASS or ClassMode.BEFORE_EACH_TEST_METHOD or ClassMode.AFTER_EACH_TEST_METHOD or ClassMode.AFTER_CLASS)`

*/

@Test**@Order(2)****@DirtiesContext(methodMode = MethodMode.BEFORE_METHOD)**

```
public void addAndGetData2InMySpringContext() {
    mySpringSetEx.addDataInExDataSet("data_2a");
    mySpringSetEx.addDataInExDataSet("data_2b");
    Set<String> dataSet2 = mySpringSetEx.getExDataSet();
    logger.debug("dataSet2 = " + dataSet2); //dataSet2 = [data_2b, data_2a]
    Assertions.assertTrue(dataSet2.size() == 2); //with success
}
```

}

4. Test d'intégration partiel

4.1. Version 1 sans @MockBean

WithMockDaoConfig.java

```
package com.mycompany.xyz.test;

import org.mockito.Mockito;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.context.annotation.Profile;

import com.mycompany.xyz.repository.RepositoryDevise;

//NB: cette classe associée au mini-profile "mock-dao"
//n'est utile que pour la classe TestServiceDeviseWithDaoMockV1
//et n'est plus nécessaire pour la V2 qui utilise @MockBean à la place de @Autowired

@Configuration
public class WithMockDaoConfig {

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

    @Bean()
    @Profile("mock-dao")
    @Primary //for overriding default spring-data-jpa dao
    public RepositoryDevise daoDeviseMock() {
        logger.info("Mocking: {}", RepositoryDevise.class);
        return Mockito.mock(RepositoryDevise.class);
    }
}
```

TestServiceDeviseWithDaoMockV1.java

```
package com.mycompany.xyz.test;

//pour assertTrue (res==5) au lieu de Assertions.assertTrue(res==5)
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.ArrayList; import java.util.List; import java.util.Optional;

import org.junit.jupiter.api.BeforeEach; import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith; import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import com.mycompany.xyz.MySpringBootApplication;
import com.mycompany.xyz.entity.Devise;
import com.mycompany.xyz.repository.RepositoryDevise;
import com.mycompany.xyz.service.ServiceDevise;
```

```

@ExtendWith(SpringExtension.class)
@SpringBootTest(classes= {MySpringBootApplication.class})
@ActiveProfiles({"embeddedDb","mock-dao"})
public class TestServiceDeviseWithDaoMockV1 {

    @Autowired
    private ServiceDevise serviceDevise; //à tester
    //(NB: ce test fonctionne avec l'implémentation ServiceDeviseV2)

    @Autowired
    private RepositoryDevise daoDeviseMock; //mock à utiliser

    @BeforeEach
    public void reInitMock() {
        //vérification que le dao injecté est bien un mock :
        assertTrue(Mockito.mockingDetails(daoDeviseMock).isMock());
        //reinitialisation du mock(de scope=Singleton par défaut) sur aspects stub et spy :
        Mockito.reset(daoDeviseMock);
    }

    @Test
    public void testRechercherDevises() {
        //préparation du mock (qui sera utilisé en arrière plan du service à tester):
        List<Devise> devises = new ArrayList<>();
        devises.add(new Devise("EUR","Euro",1.0));
        devises.add(new Devise("USD","Dollar",1.1));
        Mockito.when(daoDeviseMock.findAll()).thenReturn(devises);
        //vérification du résultat du service
        List<Devise> listeDevises = serviceDevise.rechercherDevises();
        System.out.println("listeDevises="+listeDevises);
        assertEquals(listeDevises.size(),2);
        //vérifier si le service a appeler 1 fois findAll() en interne sur le dao:
        Mockito.verify(daoDeviseMock, Mockito.times(1)).findAll();
    }

    @Test
    public void testRechercherDeviseParCode() {
        //préparation du mock (qui sera utilisé en arrière plan du service à tester):
        Devise d = new Devise("Ms","Monnaie de singe",1234.567);
        Mockito.when(daoDeviseMock.findById("Ms")).thenReturn(Optional.of(d));
        //vérification du résultat du service
        Devise deviseRemontee = serviceDevise.rechercherDeviseParCode("Ms");
        System.out.println("deviseRemontee="+deviseRemontee);
        assertEquals(deviseRemontee.getNom(),"Monnaie de singe");
        //vérifier si le service a appeler 1 fois findById() en interne sur le dao:
        Mockito.verify(daoDeviseMock, Mockito.times(1)).findById(Mockito.anyString());
    }
}

```

4.2. Version 2 avec @MockBean

TestServiceDeviseWithDaoMockV2.java

```
...
import org.springframework.boot.test.mock.mockito.MockBean;
...

@ExtendWith(SpringExtension.class)
@SpringBootTest(classes= {MySpringBootApplication.class})
@ActiveProfiles({"embeddedDb"}) //plus besoin du mini profile "mock-dao"
    //car utilisation de @MockBean dans cette V2
public class TestServiceDeviseWithDaoMockV2 {
    private static Logger logger = LoggerFactory.getLogger(TestServiceDeviseWithDaoMockV2.class);

    @Autowired
    private ServiceDevise serviceDevise; //à tester

    @MockBean
    /* @MockBean pour demander à "Spring+Mockito" de :
     * - créer un Mock de l'interface
     * - faire en sorte que ce Mock remplace le composant habituel
     *   (un peu comme WithMockDaoConfig avec @Primary)
     * - INJECTER PARTOUT (ici et dans ServiceDeviseV2) un Mock de l'interface
     *   plutôt que le véritable composant spring
     */
    private RepositoryDevise daoDeviseMock; //mock à utiliser

    @BeforeEach
    public void reInitMock() {
        //vérification que le dao injecté est bien un mock
        assertTrue(Mockito.mockingDetails(daoDeviseMock).isMock());
        //reinitialisation du mock(de scope=Singleton par défaut) sur aspects stub et spy
        Mockito.reset(daoDeviseMock);
    }

    @Test
    public void testRechercherDevises() {
        //préparation du mock (qui sera utilisé en arrière plan du service à tester):
        List<Devise> devises = new ArrayList<>();
        devises.add(new Devise("EUR","Euro",1.0));
        devises.add(new Devise("USD","Dollar",1.1));
        Mockito.when(daoDeviseMock.findAll()).thenReturn(devises);
        //vérification du résultat du service
        List<Devise> listeDevises = serviceDevise.rechercherDevises();
        System.out.println("listeDevises="+listeDevises);
        assertEquals(listeDevises.size(),2);
        //vérifier si le service a appeler 1 fois findAll() en interne sur le dao:
        Mockito.verify(daoDeviseMock, Mockito.times(1)).findAll();
    }
    ...
}
```

5. Optimisation sur la partie à charger et tester

5.1. Via configuration adéquate

ServiceAndDaoConfig.java (à mettre dans un package externe ou bien avec un profile)

```
package com.mycompany.partial_config;

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@Configuration
@EnableAutoConfiguration
@ComponentScan(basePackages = { "com.mycompany.xyz.service" })
@EnableJpaRepositories(basePackages = { "com.mycompany.xyz.repository" })
@EntityScan(basePackages = { "com.mycompany.xyz.entity" })
public class ServiceAndDaoConfig {

}

/*
Usage:
@ExtendWith(SpringExtension.class)
//{@SpringBootTest
//{@SpringBootTest(classes= {MySpringBootApplication.class})
@SpringBootTest(classes= {ServiceAndDaoConfig.class})
in order to load "DAO + Service" components only in service test
no need of "RestController" components for internal business service tests
ServiceAndDaoConfig is more light than all MySpringBootApplication .
*/
```

5.2. @DataJpaTest

```
@ExtendWith(SpringExtension.class) //si junit5/jupiter
//{@SpringBootTest(classes= {AppliSpringApplication.class}) //même config que classe avec main()
@DataJpaTest //better of SpringBootTest for dao testing if use of spring-data-jpa extension
public class TestCompteDao {
...
}
```

6. Test "end-to-end" sur backend

Voir fin du chapitre "WS-REST"

====

ANNEXES

XIII - Annexe – Selon versions (ajustements)

1. Migration de Spring 5 vers Spring 6

1.1. Changement de dépendances (pom.xml)

Souvent indirectement via Spring-boot .

Dépendances directes et explicites pour Spring6 sans SpringBoot :

```

<properties>
    <java.version>17</java.version>
    <spring.version>6.1.2</spring.version>
    <junit.jupiter.version>5.10.1</junit.jupiter.version>
</properties>
...
<dependency>
    <groupId>jakarta.inject</groupId>
    <artifactId>jakarta.inject-api</artifactId>
    <version>2.0.1</version>
</dependency> <!-- pour que Spring puisse interpréter @Inject comme @Autowired -->
<dependency>
    <groupId>jakarta.servlet</groupId>
    <artifactId>jakarta.servlet-api</artifactId>
    <version>6.0.0</version>
    <scope>provided</scope> <!-- provided par tomcat après déploiement .war -->
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.4.1.Final</version>
</dependency> <!-- et indirectement jpa en version jakarta -->
<dependency>
    <groupId>jakarta.annotation</groupId>
    <artifactId>jakarta.annotation-api</artifactId> <!-- @PostConstruct -->
    <version>2.1.1</version>
</dependency>
<dependency>
    <groupId>jakarta.servlet.jsp.jstl</groupId>
    <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
    <version>3.0.0</version>
</dependency>
<dependency>
    <groupId>org.glassfish.web</groupId>
    <artifactId>jakarta.servlet.jsp.jstl</artifactId>
    <version>3.0.0</version>
</dependency>
<!-- springdoc-openapi-ui pour spring5/springBoot2 ,
     springdoc-openapi-starter-webmvc-ui pour spring6/springBoot3 -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

```

```
<version>2.3.0</version>
</dependency>
```

1.2. Changement de packages

<i>Spring5 et JEE <=8</i>	<i>Spring 6 et JEE >=9</i>
javax.persistence.*	jakarta.persistence.*
javax.servlet.http	jakarta.servlet.http
javax.annotation	jakarta.annotation
...	

Exemples (spring 6) :

```
import jakarta.annotation.PostConstruct;
import jakarta.inject.Inject;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.*
```

Entête de l'éventuel META-INF/persistence.xml en version 3.0 :

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
    version="3.0">
```

taglib de jstl en version jarkarta :

```
<%@ taglib prefix="c" uri="jakarta.tags.core"%>
```

1.3. Continuité vis à vis des évolutions de SpringSecurity

Du coté de SpringSecurity , Encore des changements entre Spring 5 et Spring 6
peu après les changements de la version 5.7 de Spring .

Changements récents pour SpringSecurity associé à Spring6/SpringBoot3 :

```
//@EnableGlobalMethodSecurity(prePostEnabled = true) for Spring5
@EnableMethodSecurity //with default prePostEnabled = true
```

```
//Spring5/SpringBoot2 : http.authorizeRequests().antMatchers("....")....
//Spring6/SpringBoot3 : http.authorizeHttpRequests( auth -> auth.requestMatchers("....")....);
```

plus de .and(), mais des lambdas imbriquées

1.4. Changements dans les paramétrages de l'auto-configuration

Dans un sous projet de type **xyz-configure**, le paramétrage de l'auto-configuration à changé entre spring 5 et spring 6.

Avec Spring5/SpringBoot2:

ancien fichier **META-INF/spring.factories**

comportant

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=org.mycontrib.mysecurity.c  
hain.autoconfigure.MySecurityAutoConfiguration
```

Avec Spring6/SpringBoot3:

Nouveau fichier

META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports

comportant

```
org.mycontrib.mysecurity.chain.autoconfigure.MySecurityAutoConfiguration
```

1.5. Nouvelles possibilités de java 17

- **Pattern matching**: if (obj instanceof String s) { ... }
- **switch expression** (avec lambda et mot clef yield)
- **record** possibles au niveau des DTOs (avec restrictions!!!)
- **Text blocs** (entre """" et """"", pratiques pour tests unitaires)

1.6. Nouvelles possibilités de Spring 6

- **GraalVM Native Images** (voir annexe détaillée)
- **Spring Observability** (<https://www.baeldung.com/spring-boot-3-observability>)

XIV - Annexe – Configuration Spring avancée

1. "java config" avancée

1.1. Démarrage souple/contextuel

@SpringBootApplication

```
public class AppliSpringWebApplication {
    public static void initActiveProfiles() {
        String activeProfilesString = System.getProperty("spring.profiles.active");
        if(activeProfilesString!=null) {
            System.out.println("spring.profiles.active is already initialized: "
                + activeProfilesString);
        } else {
            //activeProfilesString="h2,init";
            activeProfilesString="h2,init,withSecurity";
            System.setProperty("spring.profiles.default", activeProfilesString);
            System.out.println("uninitialized spring.profiles.active" +
                " is now set to this default value: " + activeProfilesString);
        }
    }
    public static void main(String[] args) {
        initActiveProfiles();//to call before run()
        SpringApplication.run(AppliSpringWebApplication.class, args);
        System.out.println("default (dev) url: http://localhost:8181/appliSpringWeb");
        //selon application.properties
    }
}
```

Un démarrage flexible/paramétrable est très important dans un contexte "devops" (par exemple dans le cadre d'un microservice "backend_springBoot" packagé dans un conteneur docker).

Dockerfile

```
FROM openjdk:17
WORKDIR /usr/app
COPY target/appliSpringWeb.jar .
EXPOSE 8181
CMD ["java","-jar","appliSpringWeb.jar" ]
```

docker image build -t ...

docker container run -p 8181:8181 -e SPRING_PROFILES_ACTIVE=withSecurity,oracle -d

L'option **-e** de "docker container run" permet de fixer la valeur d'une variable d'environnement au moment du lancement/démarrage du conteneur docker. Et la variable d'environnement **SPRING_PROFILES_ACTIVE** (comme la propriété système `-Dspring.profiles.active`) est fondamentale pour paramétriser la liste des profils spring à activer au démarrage de l'application.

Autres possibilités autour des profiles (en mode "SpringBoot")

application.properties

```
...  
spring.profiles.active=dev,reInit  
...
```

pour décider s'il faut utiliser ou pas les fichiers annexes *application-dev.properties* etc...

+ éventuel filtrage de resources maven avec placeholder **@spring.profiles.active@** à la place de dev,reInit .

Groupe de profils :

application.properties

```
...  
spring.profiles.group.PRODUCTION=prod1,prod2  
spring.profiles.group.DEV=dev,reInit  
spring.profiles.active=DEV  
...
```

Ainsi en activant le groupe de profils "dev" , on active à la fois les profils dev1 et reInit .

1.2. Chargement automatique d'un paquet de propriétés dans un objet java (avec éventuels sous objets)

Au lieu de charger en mémoire plein de petites propriétés complémentaires avec `@Value("$xx.yy.property-name")` on peut charger d'un seul coup toute une arborescence de propriétés au sein d'un objet java via l'annotation `@ConfigurationProperties`.

Cette méthode comporte elle même différentes variantes.

La principale variante est la suivante :

```
package org.mygeneric.abc.properties;
@ConfigurationProperties(prefix = "xy")
public class XyProperties {
    private String p1;
    private Boolean p2;
    private ZzProperties zz ;
    //+ get/set et constructeur(s)
}
```

```
package org.mygeneric.abc.properties;
//sub level of properties:
public class ZzProperties {
    private String pa;
    private String pb;
    //+ get/set et constructeur(s)
}
```

application.properties

```
....
xy.p1=valeur1
xy.p2=true
xy.zz.pa=valeurA
xy.zz.pb=valeurB
```

ou bien

application.yml

```
....
xy:
  p1: valeur1
  p2: true
  zz:
    pa: valeurA
    pb: valeurB
```

Exemple d'utilisation d'un bean de "properties" au sein d'une classe de configuration :

```
package org.mygeneric.abc.autoconfigure;

@Configuration
@ConfigurationPropertiesScan("org.mygeneric.abc.properties")
public class MyAbcAutoConfiguration {

    @Autowired(required = false)
    public XyProperties xyProperties;

    @Bean
    public Prefixeur monAbc() {
        if(xyProperties!=null && xyProperties.getZz() != null) {
            return new Abc(xyProperties.getZz().getPa(),...);
        } else {
            return new Abc(); //par défaut
        }
    }
}
```

L'annotation **@ConfigurationPropertiesScan("org.mygeneric.abc.properties")** permet de préciser les packages à scanner pour trouver des classes avec **@ConfigurationProperties** servant à charger en mémoire des parties de **application.properties** ou **application.yml** sous forme de composant java (injectable via **@Autowired** ou autre).

1.3. **@Primary (version principale/prioritaire)**

NB : L'annotation **@Primary** peut être placée à coté de **@Bean** (ou **@Component**) pour désigner la version principale/prioritaire qui sera injectée en cas d'ambiguïté apparente (**@Autowired** ou autre).

1.4. "dataSource" secondaire (multiple)

src/main/resources/application.properties

```
...
#spring.xyz.datasource.url=jdbc:h2:mem:xyzDb
spring.xyz.datasource.url=jdbc:h2:~/xyzDb
spring.xyz.datasource.username=sa
spring.xyz.datasource.password=
...
```

@Configuration

```
public class xyzConfig {

    /*
     * NB: in application.properties
     * NOT spring.datasource.url=jdbc:h2:mem:xyzDb
     * BUT spring.xyz.datasource.url=jdbc:h2:~/xyzDb
     * and spring.xyz.datasource.username=sa
     *      spring.xyz.datasource.password=
     */
    @Bean @Qualifier("xyz")
    @ConfigurationProperties("spring.xyz.datasource")
    public DataSourceProperties xyzDataSourceProperties() {
        return new DataSourceProperties();
    }
}
```

```
@Bean(name="xyzDataSource") @Qualifier("xyz")
public DataSource xyzDataSource(@Qualifier("xyz") DataSourceProperties
                                xyzDataSourceProperties) {
    return xyzDataSourceProperties
        .initializeDataSourceBuilder()
        .build();
}
```

```
@Bean(name="xyzTransactionManager") @Qualifier("xyz")
public PlatformTransactionManager xyzTransactionManager(
    @Qualifier("xyz") DataSource xyzDataSource) {
    return new DataSourceTransactionManager(xyzDataSource); //ou autre (ex : JPA)
}
```

et quelquefois (selon le contexte)

```
//@....(dataSourceRef = "xyzDataSource",
//       transactionManagerRef = "xyzTransactionManager")
```

exemples:

```
@Configuration
@EnableJpaRepositories(
    entityManagerFactoryRef = "xyzEntityManager",
    transactionManagerRef = "xyzTransactionManager",
    basePackages = {"tp.appliSpring.dao"}
)
public class MySpringDataJpaConfig { ...}
```

```
@EnableJdbcRepositories(
    basePackages = "tp.appliSpring.dao",
    transactionManagerRef = "xyzTransactionManager",
    jdbcOperationsRef = "xyzJdbcNamedParameterJdbcTemplate"
)
public class MySpringDataJdbcConfig { ...}
```

1.5. ApplicationContextAware

NB : L'interface **ApplicationContextAware** peut éventuellement être implémentée par une classe de composant Spring et elle permet alors à ce composant d'accéder au contexte spring (de manière à éventuellement appeler dessus .getBean("...") ou autre par la suite).

```
@Component
public class MySpringAwareBean implements ApplicationContextAware {
    private static ApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        ApplicationContextProvider.applicationContext = applicationContext;
    }
}
```

NB : alternative (à peu près équivalente) avec @Autowired de type ApplicationContext

```
@Component
public class MyBean {

    @Autowired
    private ApplicationContext applicationContext;
}
```

2. Configuration avancée pour SpringBoot

2.1. Configuration conditionnelle intelligente

Annotations que l'on peut ajouter à coté de @Bean :

@ConditionalOnBean(name="otherBeanNameThatMustExist")

@ConditionalOnMissingBean

//pour configurer une nouvelle instance de Bean (specific type , specific name via methodName)
que si ce bean n'existe pas encore

@ConditionalOnClass(name="com.sample.Dummy")

//pour configurer un Bean **que si une classe est trouvée dans le classpath**

@ConditionalOnMissingClass(value={"com.sample.Dummy"})

//pour configurer un Bean **que si une classe n'est pas trouvée dans le classpath**

@ConditionalOnWebApplication et @ConditionalOnNotWebApplication

@ConditionalOnResource(resources={"classpath:application.properties"})

//pour configurer un Bean **que si une ressource est trouvée dans le classpath**

@ConditionalOnResource(resources={"file:///e:/doc/data.txt"})

@ConditionalOnJava(value=JavaVersion.SEVEN,range=Range.OLDER_THAN)

//pour configurer un Bean **que si version de java < 7**

@ConditionalOnProperty(name="test.property1", havingValue="A")

//pour configurer un Bean **que si la propriété "test.property1" existe dans l'environnement et vaut "A"**

@ConditionalOnProperty(name="test.property2")

//pour configurer un Bean **que si la propriété "test.property2" existe dans l'environnement et est différente de "false"**

@ConditionalOnJndi(value={"jndiName1", "jndiName2"})

//pour configurer un Bean **que si au moins un nom logique JNDI est trouvé dans InitialContext**

...

C'est ce genre de configuration automatique et intelligente qui est automatiquement activée (de manière implicite) en mode "spring-boot-starter-..." .

2.2. auto-configuration

Dans un sous projet de type **xyz-configure** , le paramétrage de l'auto-configuration à changé entre spring 5 et spring 6.

Avec Spring5/SpringBoot2:

ancien fichier **META-INF/spring.factories**

comportant

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=org.mycontrib.mysecurity.c  
hain.autoconfigure.MySecurityAutoConfiguration
```

Avec Spring6/SpringBoot3:

Nouveau fichier

META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports

comportant

```
org.mycontrib.mysecurity.chain.autoconfigure.MySecurityAutoConfiguration
```

XV - Annexe – Spring-MVC (JSP et Thymeleaf)

1. Spring-MVC avec pages JSP

1.1. Dépendances maven (SpringMvc + JSP)

Dépendances maven nécessaires (en intégration moderne "spring-boot"):

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

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

et (si vues de type ".jsp")

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>jakarta.servlet.jsp.jstl</groupId>
    <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
</dependency>
<dependency>
    <groupId>org.glassfish.web</groupId>
    <artifactId>jakarta.servlet.jsp.jstl</artifactId>
</dependency>
<!-- ou ancien équivalent spring5/springBoot2/jee --&gt;
<!-- &lt;dependency&gt;
    &lt;groupId&gt;javax.servlet&lt;/groupId&gt;
    &lt;artifactId&gt;jstl&lt;/artifactId&gt;
&lt;/dependency&gt; --&gt;</pre>

```

1.2. Configuration en version ".jsp":

src/main/resources/application.properties

```
server.servlet.context-path=/myMvcSpringBootApp
server.port=8080
#spring.mvc.view.prefix=/WEB-INF/view/
spring.mvc.view.prefix=/jsp/
spring.mvc.view.suffix=.jsp
```

Avec cette configuration , un **return "xy"** d'un contrôleur déclenchera l'affichage de la page **/jsp/xy.jsp** et selon la structure du projet , le répertoire **/jsp** sera placé dans **src/main/resources/META-INF/resources** ou ailleurs .

1.3. Exemple élémentaire (SpringMvc + JSP):

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

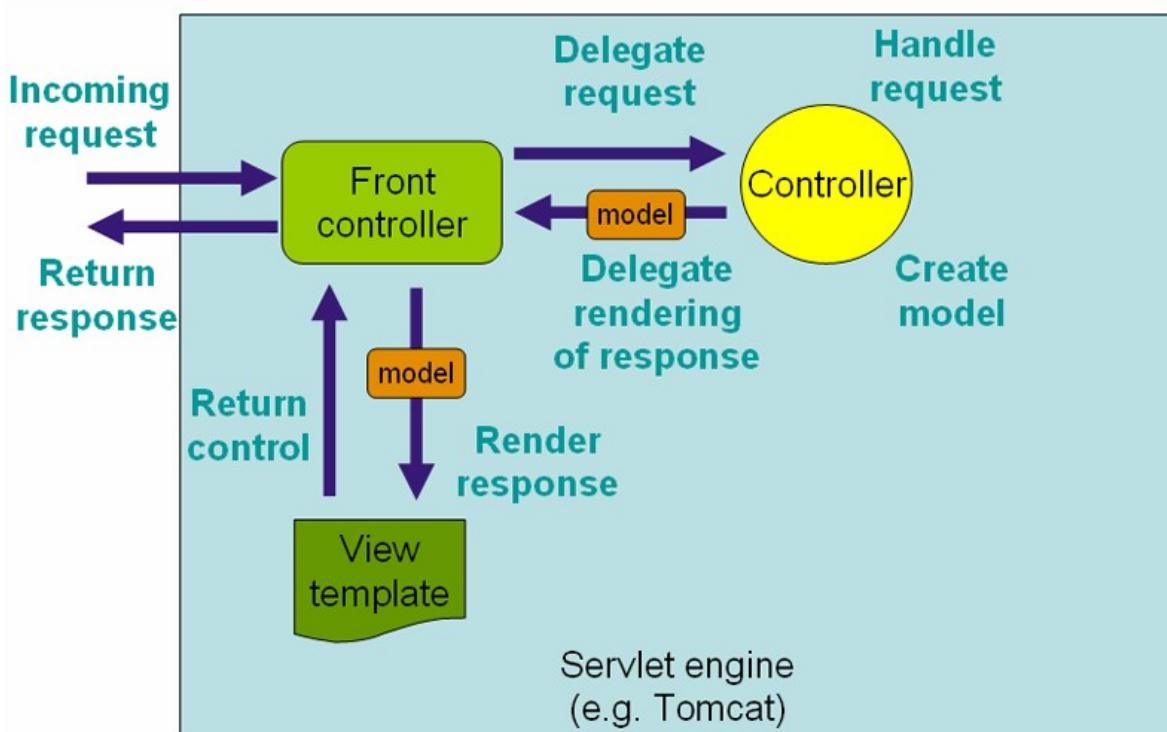
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public String helloWorld(Model model) {
        model.addAttribute("message", "Hello World!");
        return "showMessage";
    }
}
```

Au niveau de `/jsp/showMessage.jsp`, l'affichage de message pourra être effectué via `${message}`.

```
<html><head><title>showMessage</title></head>
<body>
    <p>message=<b>${message}</b></p>
</body></html>
```

Rappel du principe de fonctionnement de SpringMvc :



2. éléments essentiels de Spring web MVC

2.1. éventuelle génération directe de la réponse HTTP

```

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller //but not "@Component" for spring web controller
@RequestMapping("/app")
public class WelcomeCtrl {

    @RequestMapping("/hello")
    @ResponseBody //si @ResponseBody , génération directe de la réponse ,
                  // sinon viewResolver (.jsp ou .html thymeleaf)
    String say_hello() {
        return "Hello World!";
    }
}

```

2.2. @RequestParam (accès aux paramètres HTTP)

conversion.jsp

```

... <form action="doConversion" method="GET_ou_POST">
    source: <select name="source">
        <c:forEach var="d" items="$allDevises">
            <option value="$d.monnaie">$d.monnaie</option>
        </c:forEach>
    </select> <br/>
    cible: <select name="cible"> ... </select> <br/>
    montant: <input name="montant" value="$montant"/> <br/>
    <input type="submit" value="convertir" /> <br/>
</form>
sommeConvertie=<b>$sommeConvertie</b> ...

```

```

@RequestMapping("/doConversion")
public String doConversion(Model model, @RequestParam(name="montant")double montant,
                           @RequestParam(name="source")String monnaieSrc,
                           @RequestParam(name="cible")String monnaieDest) {
    ....
    model.addAttribute("sommeConvertie",
                      gestionDevises.convertir(montant, monnaieSrc, monnaieDest));
    return "conversion";
}

```

2.3. @ModelAttribute

Pour spécifier un attribut du modèle on peut appeler ***model.addAttribute("attrName", attrVal);*** au sein d'une méthode préfixée par **@RequestMapping** .

Une autre solution consiste à coder une méthode **addXyModelAttribute()** préfixée par **@ModelAttribute("attrName")**.

Exemple :

```
@ModelAttribute("conv")
public ConversionForm addConvAttributeInModel() {
    return new ConversionForm();
}
```

Le framework "spring mvc" va alors appeler automatiquement (*) toutes les méthodes préfixées par **@ModelAttribute** pour initialiser certains attributs du modèle avant de déclencher les méthodes préfixées par **@RequestMapping**.

L'appel n'est effectué que pour initialiser la valeur d'un attribut n'existant pas encore (pas d'écrasement des valeurs en session ni des valeurs saisies via <form:form>)

Une méthode préfixée par **@ModelAttribute** peut éventuellement avoir un paramètre préfixé par **@RequestParam(name="numCli", required=true_or_false)** mais elle n'a pas le droit de retourner une valeur "null" pour un attribut du modèle .

Variante syntaxique (en void et avec model) pour de multiples initialisations :

```
@ModelAttribute
public void addAttributesInModel(Model model){
    model.addAttribute("xx", new Cxx());
    model.addAttribute("yy", new Cyy());
}
```

Autre Exemple :

```
@Controller //but not "@Component" for spring web controller
//@Scope(value="singleton")//by default
@RequestMapping("/devises")
public class DeviseListCtrl {

    @Autowired //ou @Inject
    private GestionDevises gestionDevises;

    private List<Devise> listeDevises = null; //cache
```

```

@PostConstruct
private void loadListeDevises(){
    if(listeDevises==null)
        listeDevises=gestionDevises.getListeDevises();
}

@ModelAttribute("allDevises")
public List<Devise> addAllDevisesAttributeInModel() {
    return listeDevises;
}

@RequestMapping("/liste")
public String toDeviseList(Model model) {
    //model.addAttribute("allDevises", listeDevises);
    return "deviseList";
}
}

```

deviseList.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>liste des devises</title>
</head>
<body>
    <h3>liste des devises (spring web mvc)</h3>
    <table border="1" >
        <tr><th>code</th><th>devise</th><th>change</th></tr>
        <c:forEach var="d" items="${allDevises}" >
            <tr><td>${d.codeDevise}</td><td>${d.monnaie}</td>
                <td>${d.DChange}</td></tr>
        </c:forEach>
    </table>
    <hr/>
    <a href="../app/to_welcome">retour page accueil</a> <br/>
</body>
</html>

```

Accès à un attribut pour effectuer une mise à jour:

```

@RequestMapping("/info")
public String toInfosClient(Model model) {
    //mise à jour du telephone du client 0L (pour le fun / la syntaxe):
    Client cli = (Client) model.asMap().get("customer");
    if(cli!=null && cli.getNumero()==0L)
        cli.setTelephone("0102030405");
    return "infosClient";
}

```

2.4. @SessionAttributes

```

@Controller
//@Scope(value="singleton")//by default
@RequestMapping("/client")
@SessionAttributes( value={"customer"} )
    //noms des "modelAttributes" qui sont EN PLUS récupérés/stockés
    // en SESSION HTTP au niveau de la page de rendu
    // --> visibles en requestScope ET en sessionScope
public class ClientCtrl {

    //NB: @SessionAttributes et @ModelAttribute sont gérés avant @RequestMapping

    @ModelAttribute("customer") //NB: cette méthode n'est pas appelée/déclenchée
        //si "customer" est déjà présent en session (et par copie) dans le modèle
    public Client addCustomerAttributeInModel() {
        return new Client(0L,null,null);
    }
}

```

Mettre fin à une session http:

```

@RequestMapping("/endSession")
public String endSession(Model model,HttpSession session, SessionStatus sessionStatus) {
    /* if(model.containsAttribute("customer"))
        model.asMap().remove("customer"); */
    session.invalidate();
    sessionStatus.setComplete();
    return "infosClient";
}

```

2.5. tags pour formulaires JSP (form:form , form:input , ...)

Spring-mvc offre une bibliothèque de tags permettant de simplifier la structuration d'une page JSP comportant un formulaire (à saisir , à valider ,).

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

Ces nouvelles balises préfixées par *form*: s'utilisent quasiment de la même façon que les balises standards HTML (path="nomPropJava" à la place de name="nomParamHttp").

La principale valeur ajoutée des balises préfixées par *form*: consiste dans les liaisons automatiques entre certaines propriétés d'un objet java et les champs d'un formulaire.

Les balises <form:input ...> , <form:select> doivent être imbriquées dans <form:form >.

La balise principale d'un formulaire <**form:form** action="actionXY" modelAttribute="beanName" method="POST" > ... <form:form> ... comporte un attribut clef **modelAttribute** qui doit correspondre à un nom de "modelAttribute" lui même associé à un **objet java comportant toutes les données du formulaire à soumettre**.

Autrement dit , form:form ne fonctionne correctement que si la classe du sous-contrôleur est structurée avec au moins un "@ModelAttribute" (existant dès le départ , pas "null") dont le type correspond à une classe souvent spécifique au formulaire (ex : "UserForm" , "OrderForm" ,) .

Exemple:

```
public class ConversionForm {
    private Double montant;
    private String monnaieSrc;
    private String monnaieDest;

    public ConversionForm(){
        monnaieSrc="dollar";
        monnaieDest="dollar"; //par défaut (dans formulaire avant saisies)
    }
    //+ get/set
}
```

```
@Controller
//@Scope(value="singleton")//by default
@RequestMapping("/devises")
public class DeviseListCtrlV2 {
    ...
    //pour modelAttribute="conv" de form:form
    @ModelAttribute("conv")
    public ConversionForm addConvAttributeInModel() {
        return new ConversionForm();
    }
    ...
}
```

L'attribut path="..." des sous balises <form:input ...> , <form:select> font alors référence aux propriétés de l'objet java (en lecture/écriture , get/set) .

NB: <form:form ...> gère (génère) automatiquement le champ caché **_csrf** attendu par **spring-security** . Exemple : <input type="hidden" name="**_csrf**" value="8df91b84-74c1-4013-bd44-ede7b00779a2" /> . Ce champ caché correspond au "*Synchronizer Token Pattern*" (que l'on retrouve dans les frameworks web concurrents "Stuts" ou "JSF") : le coté serveur compare la valeur d'un jeton aléatoire stockée en session http avec celle stockée dans un champ caché et refuse de gérer la requête "re-postée" si la comparaison n'est pas réussie.

D'autre part , le terme **CSRF** (signifiant "Cross Site Request Forgery" correspond à un éventuel problème de sécurité : un site "malveillant" (utilisé en parallèle au sein d'un navigateur) déclenche automatiquement (via javascript ou autre) des requêtes non voulues (ex : virement monétaire) en utilisant le contexte d'un site à priori de confiance (mais pas assez protégé) .

Avec <form> (au lieu de <form:form>) , il faut insérer nous même le champ suivant au sein du formulaire d'une page ".jsp" :

```
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
```

conversionV2.jsp

```
<form:form action="doConversion" modelAttribute="conv" method="POST">
    source: <form:select path="monnaieSrc" >
        <form:options items="${allDevises}" itemLabel="monnaie"
itemValue="monnaie"/>
    </form:select> <br/>
    cible: <form:select path="monnaieDest" >
        <form:options items="${allDevises}" itemLabel="monnaie" itemValue="monnaie"/>
    </form:select> <br/>
    montant: <form:input path="montant" />
    <form:errors path="montant" cssClass="error"/><br/>
    <input type="submit" value="convertir" /> <br/>
</form:form>
sommeConvertie=<b>${sommeConvertie}</b>
```

conversion de devises

source:

cible:

montant:

sommeConvertie=37.5

Finalement , au sein du contrôleur , la méthode déclenchée par le formulaire peut s'écrire de la façon suivante:

```
@RequestMapping("/doConversion")
public String doConversion(Model model,@ModelAttribute("conv") ConversionForm conv ) {
    model.addAttribute("sommeConvertie",
        gestionDevises.convertir(conv.getMontant(),
        conv.getMonnaieSrc(), conv.getMonnaieDest()));
    return "conversionV2";
}
```

2.6. validation lors de la soumission d'un formulaire

Rappel: la classe de l'objet utilisé en tant que "modelAttribute" au niveau d'un formulaire peut comporter des annotations @Min , @Max , @Size , @NotEmpty , ... de l'api normalisée javax.validation .

Exemples :

```
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;

public class ConversionForm {

    @Min(value=0)
    @Max(value=999999)
    private Double montant;
    ...
}
```

```
import javax.validation.constraints.Size;
import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.NotEmpty;

public class Client {
    private Long numero;  private String nom;      private String prenom;
    @NotEmpty(message = "Please enter your address.")
    @Size(min = 4, max = 128, message = "Your address must between 4 and 128 characters")
    private String adresse;
    private String telephone;

    @NotEmpty
    @Email
    private String email;
    ...
}
```

Il suffit en suite d'ajouter **@Valid** au niveau du paramètre de la méthode associée à la soumission du formulaire pour que spring-mvc tienne compte des contraintes de validation.

D'autre part, le paramètre (facultatif mais conseillé) de type "**BindingResult**" permet de gérer finement les cas d'erreur de validation :

```
@RequestMapping("/doConversion")
public String doConversion(Model model,
                           @ModelAttribute("conv") @Valid ConversionForm conv ,
                           BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        // form validation error
        System.out.println("form validation error: " + bindingResult.toString());
    } else {
```

```
// form input is ok*/
model.addAttribute("sommeConvertie", gestionDevises.convertir(conv.getMontant(),
                                                        conv.getMonnaieSrc(), conv.getMonnaieDest()));

}

return "conversionV2";
}
```

conversion de devises

source: ▼

cible: ▼

montant: doit être plus grand que 0

sommeConvertie=

[retour page accueil](#)

numero: 0

nom:

prenom:

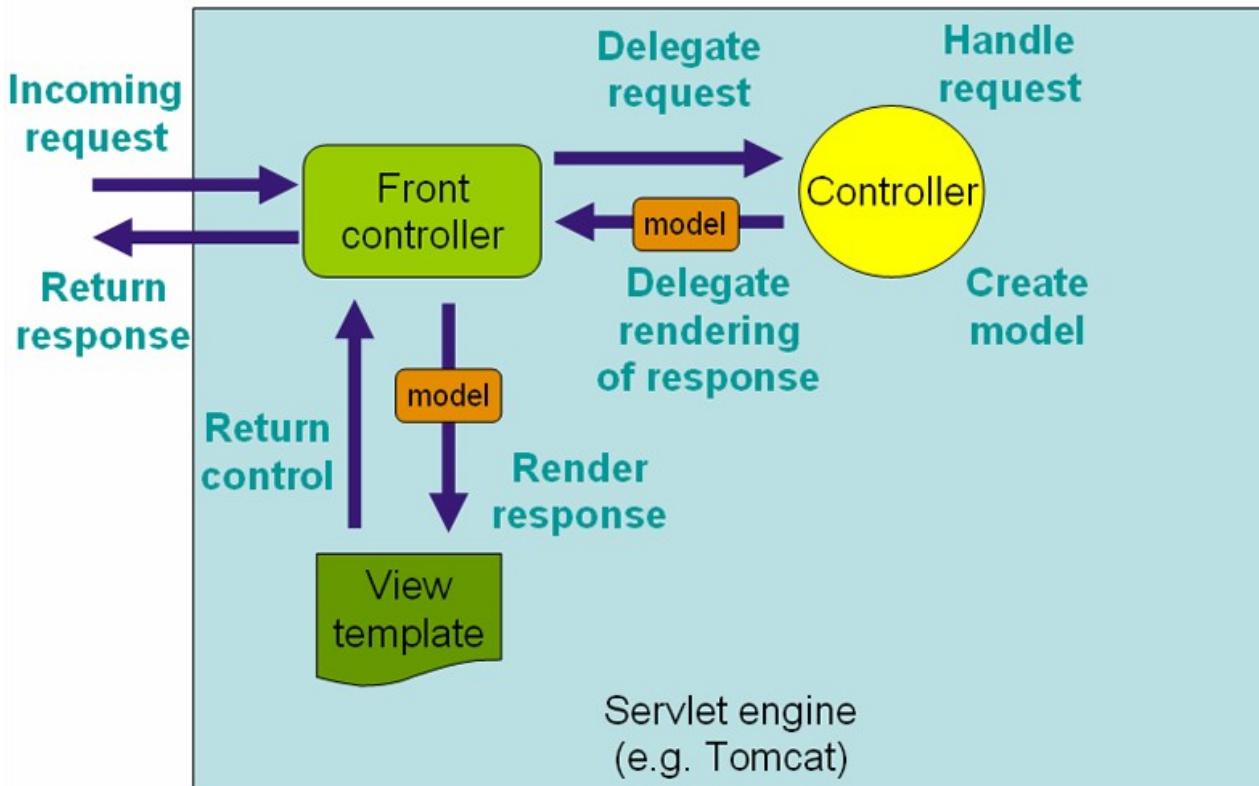
adresse: Your address must between 4 and 128 characters

telephone:

email: Adresse email mal formée

3. Spring-Mvc avec Thymeleaf

3.1. Vues en version Thymeleaf



Les vues peuvent être en version "thymeleaf" plutôt que "JSP"

3.2. Spring-mvc avec Thymeleaf

La technologie "Thymeleaf" est une alternative intéressante vis à vis des pages JSP et qui offre les avantages suivants :

- **syntaxe plus développée** (plus concise , plus expressive , plus sophistiquée)
- **meilleures possibilités/fonctionnalités pour la mise en page** (héritage de layout , ...)
- technologie assez souvent utilisée avec SpringMvc et SpringBoot

Rappel des dépendances maven nécessaires :

```

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

<dependency> <!-- pour @Max , ... @Valid -->
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

<dependency>
    <groupId>nz.net.ultraq.thymeleaf</groupId>
    <artifactId>thymeleaf-layout-dialect</artifactId>
</dependency>

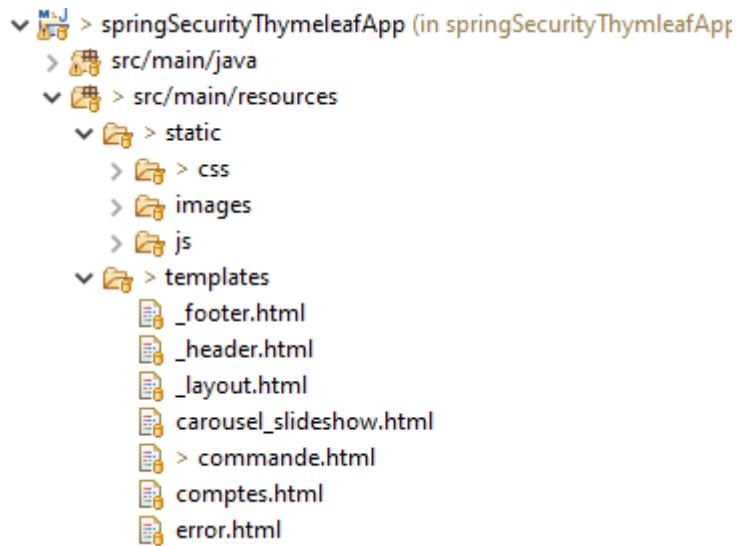
<!--
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity5</artifactId>
</dependency>
-->

```

Sans configuration spécifique dans application.properties le répertoire prévu pour accueillir les templates de **thymeleaf** est **src/main/resources/templates** .

Sachant que les fichiers annexes ".css" , ".js" , ... sont à ranger dans **src/main/resources/static**.



Il n'y a pas de différence notable dans l'écriture des contrôleurs (JSP ou Thymeleaf : peu importe).

3.3. "Hello world" avec Spring-Mvc et Thymeleaf

src/main/resources/static/index.html

```
<a href="site/app/hello-world-th">hello-world-th (via Spring Mvc et thymeleaf)</a> <br/>
```

AppCtrl.java

```
package tp.appliSpring.web.controller;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/site/app")
public class AppCtrl {

    @RequestMapping("/hello-world-th")
    public String helloWorld(Model model) {
        model.addAttribute("message", "Hello World!");
        return "showMessage"; //aiguiller sur la vue "showMessage"
    }
}
```

src/main/resources/templates/showMessage.html

```
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>showMessage</title>
</head>
<body>
    message: <span th:utext="${message}"></span>
</body>
</html>
```

Résultat :

```
message: Hello World!
```

3.4. Templates thymeleaf avec layout

Voici quelques exemples de "vues/templates" basés sur la technologie "Thymeleaf" :

_header.html

```
<header xmlns:th="http://www.thymeleaf.org" th:fragment="_header">
<div style="width:100%;font-size:36px;line-height:48px;background-color:navy;color:white">
    My SpringMVC Thymeleaf Application</div> </header>
```

_footer.html

```
<footer xmlns:th="http://www.thymeleaf.org" th:fragment="_footer">
<div style="background-color:navy;width:100%;color:white">
    Mon pied de page ... <a th:href="@{/to-welcome}" style="color:yellow" >welcome</a>
</div> </footer>
```

NB :

- **th:href="@{/to-welcome}" au sens th:href="@{controller_requestMapping}"**
- les *sous fichiers* _header.html et _footer.html seront **inclus** dans _layout.html via **th:replace="..."**

Le fichier *_layout.html* suivant correspond à un template/**modèle commun/générique de mise en page**. La plupart des pages ordinaires de l'application reprendront (par héritage) la structure de *_layout.html*.

Le contenu des zones identifiées par **layout:fragment="nomLogiqueFragment"** pourront si besoin est redéfinies/remplacées au sein des futures pages basées sur ce template :

_layout.html

```
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout" >
<head>
    <meta charset="UTF-8" />
    <title layout:fragment="title" th:utext="${title}"></title>
    <link rel="stylesheet" type="text/css" th:href="@{/css/bootstrap.min.css}"/>
    <link rel="stylesheet" type="text/css" th:href="@{/css/styles.css}"/>
</head>
<body>
    <div class="container-fluid">
        <div th:replace="_header"></div>
        <div layout:fragment="content">
            default content from _layout.html (to override)
        </div>
        <div th:replace="_footer"></div>
    </div><!-- end of bootstrap css container-fluid -->
</body>
</html>
```

Le fichier *welcome.html* suivant est basé sur le modèle générique *_layout.html* via le lien d'héritage / de composition **layout:decorate="~{ _layout }"**.

Au sein de *welcome.html*, tout le contenu imbriqué entre début et fin de la balise marquée via **layout:fragment="content"** va automatiquement remplacer le texte *default content from _layout.html (to override)* qui était encadré par la même nom logique de fragment au sein de *_layout.html*.

Le rendu globalement fabriqué par Thymeleaf sera ainsi une page HTML complète ayant comme structure celle de _layout.html (et donc avec _header et _footer par défaut) et dont le fragment "content" aura été redéfini avec un contenu spécifique à welcome.html.

welcome.html

```
<div xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/extras/spring-security"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorate="~{ _layout }" layout:fragment="content">
    <h1>Welcome Thymeleaf (public part)</h1>
    <p>message=<b><span th:utext="${message}"></span></b></p>
    <hr/> ... /div>
```

My SpringMVC Thymeleaf Application

Welcome Thymeleaf (public part)

message=bienvenu(e)

nouveau client
welcome-authenticated with loginSpringSecurity.html automatic hook (client or admin)
update commande
exemple ajax
exemple carousel

fin de session / deconnexion
num session http/jee= F4386246590C8C7FD57CC251B4AB40C0

valid accounts (dev): customer(1, pwd1), customer(2, pwd2), admin(superAdmin, 007)
Mon pied de page ... welcome

Exemple de formulaire ultra simple (élémentaire) avec thymeleaf :

```
...
<form th:action="@{/site/compte/verifLogin}" method="POST">
    numClient : <input name="numClient" type="text" /> <br/>
    <input type="submit" value="identification client banque" /> <br/>
    <input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}"/>
</form> ...
```

en liaison avec

```
@Controller
@RequestMapping("/site/compte")
public class CompteController {
    @RequestMapping("/verifLogin")
    public String verifLogin(Model model,
        @RequestParam(name="numClient") Long numClient) { ... } }
```

NB: (@RequestParam(name="numClient", required = false) Long numClient) {
 if(numClient==null) {} else {....}
 }

NB2: monCalcul(Model model, @RequestParam(name="val",defaultValue = "0") double val)
 peut être pratique pour récupérer (la première fois) une valeur par défaut si l'on est pas encore passé
 par un petit formulaire de saisie

Exemple de formulaire simple avec thymeleaf :

```
<div xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorate="~{_layout}" layout:fragment="content">
    <h3>nouveau (client banque)</h3>
    <hr/>
    <form th:action="@{/nouveauClient}"
          th:object="${client}" method="POST">
        numero : <input th:field="*{numero}" type="text" /> <br/>
        password : <input th:field="*{password}" type="text" /> <br/>
        nom : <input th:field="*{nom}" type="text" /> <br/>
        prenom : <input th:field="*{prenom}" type="text" /> <br/>
        <input type="submit" value="enregistrer nouveau client" /> <br/>
    </form>
</div>
```

pour déclencher :

```
@RequestMapping(value="/nouveauClient")
public String nouveauClient(Model model,
    @ModelAttribute("client") Client client) {
    //client avec propriétés .getNom() , .getPrenom() , ...
}
```

Rappels sur la gestion d'une session Spring-Mvc :

Pour spécifier un attribut du modèle on peut appeler **model.addAttribute("attrName", attrVal);** au sein d'une méthode préfixée par **@RequestMapping**.

Une autre solution consiste à coder une méthode **addXyModelAttribute()** préfixée par **@ModelAttribute("attrName")**.

Exemple :

```
@ModelAttribute("client")
public Client addClientAttributeInModel() {
    return new Client();
}
```

Le framework "spring mvc" va alors appeler automatiquement (*) toutes les méthodes préfixées par **@ModelAttribute** pour initialiser certains attributs du modèle avant de déclencher les méthodes préfixées par **@RequestMapping**.

L'appel n'est effectué que pour initialiser la valeur d'un attribut n'existant pas encore (pas d'écrasement des valeurs en session ni des valeurs saisies via <form:form>)

D'autre part, sur une classe de type **@Controller** on peut placer l'annotation

@SessionAttributes(value={"client", "caddy", "..."})

de manière à préciser les noms des "modelAttributes" qui sont EN PLUS récupérés/stockés en SESSION HTTP au niveau de la page de rendu --> visibles en **requestScope** ET en **sessionScope**

NB: au sein d'un template "thymeleaf" , on peut accéder aux attributs stockés en session via la syntaxe **\${#ctx.session.attributeName}**

Exemple :

```
<input type="text" name="numClient" th:value="${#ctx.session.numClient}" />
ou bien ${#ctx.session.client.num} selon les structures de données choisies
```

Logout avec Spring-Mvc (jsp ou thymeleaf) :

```
@RequestMapping("/logout")
public String userLogout(Model model,
                        HttpSession httpSession,
                        SessionStatus sessionStatus) {
    httpSession.invalidate();
    sessionStatus.setComplete();
    model.addAttribute("message", "session terminée");
    model.addAttribute("title", "welcome");
    return "welcome";
```

```
}
```

Exemple partiel de formulaire complexe avec thymeleaf :

```
<div xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorate="~{__layout__}" layout:fragment="content">
<script>    function onDelete(idToDelete,bToDelete){ /* .... */ } </script>
<h3>commande</h3>
<hr/>
<form th:action="@{/update-commande}" th:object="${cmdeF}" method="POST">
    numero : <label th:text="*{cmde.numero}"></label>
        <input th:field="*{cmde.numero}" type="hidden" /><br/>
    sDate : <input th:field="*{cmde.sDate}" type="text" /> <br/>
    id (client) : <label th:text="*{cmde.client.id}"></label>
        <input th:field="*{cmde.client.id}" type="hidden" /><br/>
    nom (client) : <input th:field="*{cmde.client.nom}" type="text"
                    th:class="*{cmde.client.nom == 'Bon' ? 'enEvidence' : ''}" /><br/>
    prenom (client): <input th:field="*{cmde.client.prenom}" type="text" /> <br/>
    <div th:each="p, rowStat : *{cmde.produits}">
        <hr/>
        ref (produit) : <label th:text="*{cmde.produits[__${rowStat.index}__].ref}"></label>
            <input th:field="*{cmde.produits[__${rowStat.index}__].ref}" type="hidden" /><br/>
        label (produit) : <input type="text" th:field="*{cmde.produits[__${rowStat.index}__].label}" />
        prix (produit) : <input type="text" th:field="*{cmde.produits[__${rowStat.index}__].prix}" />
        : <input type="checkbox" value="delete"
            th:onclick="onDelete(' + *{cmde.produits[__${rowStat.index}__].ref} + ',this.checked)" /> delete
        </div>
        <hr/>
        nb new product to add : <input th:field="*{prodActions.nbNew}" type="text" /> <br/>
        id of product(s) to delete :<input th:field="*{prodActions.idsToDelete}" class="RedCssClass" type="text" /> <br/>
        <input type="submit" value="update commande" /> <br/>
    </div>
</div>
```

numero : 1

sDate :

id (client) : 1

nom (client) :

prenom (client):

ref (produit) : 1

label (produit) : prix (produit) : : delete

ref (produit) : 2

label (produit) : prix (produit) : : delete

ref (produit) : 3

label (produit) : prix (produit) : : delete

nb new product to add :

id of product(s) to delete :

3.5. Principales syntaxes pour thymeleaf

Affichage des erreurs de validations (si `@Valid` du coté contrôleur) :

```
<form th:action="@{/site/....}" th:object="${....}">
    <label>montant:</label>
    <input type="text" name="nom" th:field="*{montant}" />
        <span th:if="#${#fields.hasErrors('montant')}" th:errorclass="error" th:errors="*{montant}">
            ...
        </span>
```

Saisie de dates (LocalDate coté java) :

```
<input type="text" name="dateDebut" th:field="*{dateDebut}" />
(ex: 2024-12-25 ou 25/12/2024)<br/>
```

Case à cocher en liaison avec un booléen :

```
<label class="simpleAlign">célibataire:</label>
<input type="checkbox" name="celibataire" th:checked="*{celibataire}" />
```

Boutons radios exclusifs :

```
<input type="radio" name="situation" th:value="CELIBATAIRE" th:field="*{situation}" />
    <span th:utext="CELIBATAIRE"></span> &nbsp; &nbsp;
<input type="radio" name="situation" th:value="EN_COUPLE" th:field="*{situation}" />
    <span th:utext="EN_COUPLE"></span>
```

Liste déroulante :

```
<select th:field="*{numProduit}">
    <option th:each="p : ${listeProduits}" th:value="${p.numero}"
           th:utext="${p.numero} - ${p.label} : ${p.prix}"></option>
</select> <br/>
```

Eventuelle boucle sur valeurs possibles d'une énumération :

```
<.... th:each="j : ${T(tp.appliSpringMvc.web.model.Jour).values()}">
    <... th:value="${j}" ...>
</...>
```

3.6. Sécurité avec SpringMvc et thymeleaf

Si thymeleaf est utilisé conjointement avec **spring-security** alors les éléments suivants peuvent être utiles :

@ControllerAdvice

```
public class ErrorCtrlAdvice {
    private static Logger logger = LoggerFactory.getLogger(ErrorController.class);

    @ExceptionHandler(Throwable.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public String exception(final Throwable throwable, final Model model) {
        logger.error("Exception during execution of SpringSecurity application", throwable);
        String errorMessage = (throwable != null ? throwable.getMessage() : "Unknown error");
        model.addAttribute("errorMessage", errorMessage);
        return "error";
    }
}
```

error.html

```
<!DOCTYPE HTML>
<div xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorate="~{_layout}"      layout:fragment="content">

    <div th:with="httpStatus=$
{T(org.springframework.http.HttpStatus).valueOf(#response.status)}">
        <h3 th:text="|${httpStatus} - ${httpStatus.reasonPhrase}|">404</h3>
        <p th:utext="${errorMessage}">Error</p>
        <a href="welcome.html" th:href="@{/to-welcome}">Back to Home Page (welcome)</a>
    </div>
</div>
```

500 INTERNAL_SERVER_ERROR - Internal Server Error

Accès refusé

[Back to Home Page \(welcome\)](#)

```
<!-- necessite thymeleaf-extras-springsecurity5_ou_6 dans pom.xml -->
<div xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/extras/spring-security"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorate="~{_layout}" layout:fragment="content">
    <h1>Welcome Thymeleaf (for Authenticated user)</h1>
    <p>message=<b><span th:utext="${message}"></span></b></p>
    <hr/>
    <div sec:authorize="isAuthenticated()">
        <h3>authenticated user </h3>
        Logged user: <span sec:authentication="name">Unknown</span> <br/>
        Roles: <span sec:authentication="principal.authorities">[]</span> <br/>
    </div>
    <hr/>
    ....
</div>
```

authenticated user

Logged user: superAdmin

Roles: [ROLE_ADMIN]

Quelques liens hypertextes pour approfondir "thymeleaf" :

- <https://www.thymeleaf.org/documentation.html>
- <https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html>
- <https://gayerie.dev/docs/spring/spring/thymeleaf.html>

XVI - Annexe – *async (reactor , webflux)*

1. ReactiveStreams" avec Reactor

Reactor est une implémentation proche des "*Reactive Streams*" de java>=9 .

La classe reactor.adapter.JdkFlowAdapter comporte des méthodes .publisherToFlowPublisher() et .flowPublisherToFlux() permettant de convertir des **Flux** de **Reactor** en Flow java9 et vice versa.

Le projet "Reactor" est en partie géré par la communauté "Spring" et est intégré dans le framework "Spring 5et6" avec les "WebFlux" .

Reactor est un projet plus récent que RxJava. Il nécessite au minimum le jdk 1.8 (comme Spring5)

URL officielle de Reactor : <https://projectreactor.io>

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.3.22.RELEASE</version>
</dependency>
```

Exemple simple **BasicApp.java**

```
package tp.main;

import java.util.ArrayList;
import org.reactivestreams.Subscriber; //version plus élaborée que java.util.concurrent.Flow.Subscriber
import org.reactivestreams.Subscription; //version plus élaborée que java.util.concurrent.Flow.Subscription
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public class BasicApp {
    private String result="";

    public static void main(String[] args) {
        helloWorldReactor();
        BasicApp basicApp = new BasicApp();
        basicApp.withOnNextOnErrorOnCompleteCallbacks();
        collectingElementsV1();
    }

    public static void helloWorldReactor() {
        //Mono is a sort of Observable stream with 0 or 1 element
        Mono<String> monoHello = Mono.just("hello world Reactor");
        //org.reactivestreams.Publisher<String> pubHello = Mono.just("hello world Reactor");
        //Mono is a sort of reactiveStreams Publisher
    }
}
```

```

        monoHello.subscribe(s -> System.out.println(s));
    }

    public void withOnNextOnErrorOnCompleteCallbacks() {
        result="";
        //Flux is a sort of Observable stream with 0 or n elements
        String[] letters = {"a", "b", "c", "d", "e", "f", "g"};
        Flux<String> fluxLetters = Flux.fromArray(letters);
        //org.reactivestreams.Publisher<String> pubLetters = Flux.fromArray(letters);
        //Flux is a sort of reactivestreams Publisher
        fluxLetters.subscribe(
            letter -> result += letter, //OnNext
            Throwable::printStackTrace, //OnError
            () -> result += "_completed" //OnCompleted
        );
        System.out.println("result="+result);//abcdefg_completed
    }

    public static void collectingElementsV1() {
        var resultElements = new ArrayList<>();

        Flux.just(1, 2, 3, 4)
            .log()
            .subscribe(/*resultElements::add*/ e -> resultElements.add(e));

        System.out.println("resultElements="+resultElements);
    }
}

```

2. WebFlux

2.1. Reactive WebClient

Voir partie "*Appel moderne/asynchrone de WS-REST avec WebClient*"
du chapitre "*WS_REST avec Spring-mvc*" .

2.2. Reactive WebController

```
@RestController
@RequestMapping("/employees")
public class EmployeeController {

    private final EmployeeCrudRepository employeeRepository;

    // + constructor...

    @GetMapping("/{id}")
    private Mono<Employee> getEmployeeById(@PathVariable String id) {
        return employeeRepository.findEmployeeById(id);
    }

    @GetMapping
    private Flux<Employee> getAllEmployees() {
        return employeeRepository.findAllEmployees();
    }

    // ...
}
```

NB : *EmployeeRepository can be any data repository that supports non-blocking reactive streams.*

Il existe pour l'instant une version réactive de Spring-Data pour mongoDB :

```
@Repository
public interface EmployeeCrudRepository
    extends ReactiveCrudRepository<Employee, String> {

    Flux<Employee> findAllByValue(String value);
    Mono<Employee> findFirstByOwner(Mono<String> owner);
}
```

avec ***spring-boot-starter-data-mongodb-reactive et ... etc*** .

Ceci-dit , la stack n'est pas du tout prête en version asynchrone dans le mode JPA/JDBC utilisé par la majorité des projets Spring-boot ...

XVII - Annexe – Spring Actuator

1. Spring-Actuator

1.1. Présentation de "spring-actuator"

L'extension standard "spring-actuator" permet de récupérer automatiquement des indications (métriques) sur le fonctionnement interne d'une application spring-boot .

Spring-actuator permet très concrètement d'intégrer dans l'application développée *tout un tas de micros-services REST techniques/annexes* permettant de *récupérer certaines métriques (variables d'environnement , consommation mémoire , ...)* .

Exemple :

`http://localhost:8181_ou_autre/my-spring-boot-app/actuator/health`

retourne `{"status":"UP"}` quand l'application fonctionne bien
ou bien `{"status":"DOWN"}` quand l'application ne fonctionne pas bien

En production/exploitation, Ceci peut être très utile pour effectuer un suivi/pilotage de l'application et surveiller son bon fonctionnement .

En développement/debug , spring-actuator peut grandement aider à trouver certains réglages permettant d'optimiser les performances de l'application .

1.2. Mise en oeuvre de spring-actuator

dans pom.xml :

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

dans **application.properties**

```
#actuators:
#management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.include=health,info,env,metrics
```

Important : * en dev seulement ,

En production , trop de mesures = application ralentie !!!

Et si besoin dans WebSecurityConfig ou ... :

```
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    ....
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/actuator/**").permitAll()
            .anyRequest().authenticated();
    }
}
```

1.3. URLs des "endpoints" de spring actuator :

http://localhost:8181_ou_autre/my-spring-boot-app/actuator

renvoie au format JSON la liste des urls précises des "endpoints" de spring-actuator :

```
{
  "_links": {
    "self": {
      "href": "http://localhost:8181/spring-boot-backend/actuator",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8181/spring-boot-backend/actuator/beans",
      "templated": false
    },
    "caches-cache": {
      "href": "http://localhost:8181/spring-boot-backend/actuator/caches/{cache}",
      "templated": true
    },
    "caches": {
      "href": "http://localhost:8181/spring-boot-backend/actuator/caches",
      "templated": false
    },
    "health": {
      "href": "http://localhost:8181/spring-boot-backend/actuator/health",
      "templated": false
    },
    "health-path": {
      "href": "http://localhost:8181/spring-boot-backend/actuator/health/{path}",
      "templated": true
    },
    "info": {
      "href": "http://localhost:8181/spring-boot-backend/actuator/info",
      "templated": false
    }
  }
}
```

```

    "templated": false
},
"conditions": {
  "href": "http://localhost:8181/spring-boot-backend/actuator/conditions",
  "templated": false
},
"configprops": {
  "href": "http://localhost:8181/spring-boot-backend/actuator/configprops",
  "templated": false
},
"env": {
  "href": "http://localhost:8181/spring-boot-backend/actuator/env",
  "templated": false
},
"env-toMatch": {
  "href": "http://localhost:8181/spring-boot-backend/actuator/env/{toMatch}",
  "templated": true
},
"loggers": {
  "href": "http://localhost:8181/spring-boot-backend/actuator/loggers",
  "templated": false
},
"loggers-name": {
  "href": "http://localhost:8181/spring-boot-backend/actuator/loggers/{name}",
  "templated": true
},
"heapdump": {
  "href": "http://localhost:8181/spring-boot-backend/actuator/heapdump",
  "templated": false
},
"threaddump": {
  "href": "http://localhost:8181/spring-boot-backend/actuator/threaddump",
  "templated": false
},
"metrics-requiredMetricName": {
  "href": "http://localhost:8181/spring-boot-backend/actuator/metrics/{requiredMetricName}",
  "templated": true
},
"metrics": {
  "href": "http://localhost:8181/spring-boot-backend/actuator/metrics",
  "templated": false
},
"scheduledtasks": {
  "href": "http://localhost:8181/spring-boot-backend/actuator/scheduledtasks",
  "templated": false
},
"mappings": {
  "href": "http://localhost:8181/spring-boot-backend/actuator/mappings",
  "templated": false
}
}
}
}

```

<http://localhost:8181/my-spring-boot-app/actuator/metrics/http.server.requests>

retourne

```
{
  "name": "http.server.requests",
  "description": null,
  "baseUnit": "seconds",
  "measurements": [

```

```
{
  "statistic": "COUNT",
  "value": 95
},
{
  "statistic": "TOTAL_TIME",
  "value": 4.648203600000001
},
{
  "statistic": "MAX",
  "value": 0.7233778
}
],
....
```

avec dans **application.properties**

```
#management.endpoint.env.show-values=WHEN_AUTHORIZED
management.endpoint.env.show-values=ALWAYS
```

<http://localhost:8181/my-spring-boot-app/actuator/env>

retourne

```
{
  "activeProfiles": [
    "embeddedDb",
    "reInit",
    "appDbSecurity"
  ],
  "propertySources": [
    {
      "name": "server.ports",
      "properties": {
        "local.server.port": {
          "value": 8181
        }
      }
    },
    {
      "name": "servletContextInitParams",
      "properties": {}
    },
    {
      "name": "systemProperties",
      "properties": {
        "sun.desktop": {
          "value": "windows"
        },
        "awt.toolkit": {
          "value": "sun.awt.windows.WToolkit"
        },
        "java.specification.version": {
          "value": "11"
        },
        "sun.cpu.isalist": {
          "value": "amd64"
        }
      }
    }
  ]
}
```

```

"sun.jnu.encoding": {
    "value": "Cp1252"
},
"java.class.path": {
    "value": "D:\\tp\\local-git-mycontrib-repositories\\env-ic-my-java-
rest-app\\target\\classes;...."
}, ...,

"java.vendor.url": {
    "value": "http://java.oracle.com/"
},
"catalina.useNaming": {
    "value": "false"
},
"user.timezone": {
    "value": "Europe/Paris"
},
"os.name": {
    "value": "Windows 10"
},
"java.vm.specification.version": {
    "value": "11"
},
"sun.java.launcher": {
    "value": "SUN_STANDARD"
},
"user.country": {
    "value": "FR"
}, ...

"JAVA_HOME": {    "value": "C:\\Program Files\\Java\\jdk-11.0.4",
    "origin": "System Environment Property \"JAVA_HOME\""
}, ...

```

et <http://localhost:8181/my-spring-boot-app/actuator/env/user.timezone>

retourne

```
{
"property": {
    "source": "systemProperties",
    "value": "Europe/Paris"
}, ...
}
```

...

1.4. Paramétrages (application.properties) de actuator/info

En ajoutant dans **app.properties**

```
#app infos for actuator/info if management.info.env.enabled=true
management.info.env.enabled=true
```

```
#info.app.MY_APP_PROP_NAME=ValeurQuiVaBien
info.app.name=spring-boot-backend
info.app.description=appli spring-boot , backend with rest-api (micro services)
```

alors l'actuator prédefini **actuator/info** renvoi

```
{
  "app": {
    "name": "spring-boot-backend",
    "description": "appli spring-boot , backend with rest-api (micro services)"
  }
}
```

1.5. Codage d'un "health indicator" spécifique

Avec dans application.properties :

```
management.endpoint.health.show-details=ALWAYS
```

et avec

```
@Component
public class MyHealthIndicator implements HealthIndicator {

    long checkSomething() {
        return 1;
    }

    @Override
    public Health health() {
        long result = checkSomething();
        if (result <= 0) {
            return Health.down().withDetail("Something Result", result).build();
        }
        return Health.up().withDetail("isXxxOk", true)
                        .withDetail("isYyyOk", true).build();
    }
}
```

```
▼ my:
  status:          "UP"
  ▼ details:
    isXxxOk:      true
    isYyyOk:      true
```

est retourné par **actuator/health** .

Autre variante possible : ... implements ReactiveHealthIndicator

1.6. SpringBootAdmin (extension de de.codecentric)

SpringBootAdmin est une extension spring de " de.codecentric" qui permet de mettre assez facilement en oeuvre un serveur de surveillance/administration de certaines applications **springBoot**.

- Dans la terminologie "spring-boot-admin", l'application spring-admin qui va surveiller les autres sera vu comme le coté **serveur**.
- Les applications ordinaires (avec WS REST et actuators) seront considérées comme le coté **"client"**.
- **Dans le mode de fonctionnement le plus simple , une instance d'une application ordinaire s'enregistre au démarrage auprès du serveur de surveillance "spring-admin" en précisant si besoin url,username,password .**
- Dans un mode de fonctionnement plus élaboré , l'application sring-admin ("serveur") peut quelquefois découvrir automatiquement certaines applications micro-services à surveiller via certains services techniques additionnels (ex : Eureka ou ...)

The screenshot shows the Spring Boot Admin dashboard. At the top, there are navigation links: Tableau de bord, Applications, Journal, À propos, and user authentication (admin, fr). Below the header, three summary metrics are displayed: APPLICATIONS (1), INSTANCES (1), and STATUT (tout est disponible). A detailed view for the single application is shown, including its name (spring-boot-application), ID (e17564762c69), and URLs for its actuator endpoints. The application status is marked as OK. In the bottom section, two tabs are visible: Info and État. The Info tab displays metadata like app name and description, and a timestamp for startup. The État tab shows the instance status as UP.

APPLICATIONS	INSTANCES	STATUT
1	1	tout est disponible

OK

✓ spring-boot-application
16m <http://LAPTOP-DDC:8181/spring-boot-backend> ⟳ ⚡

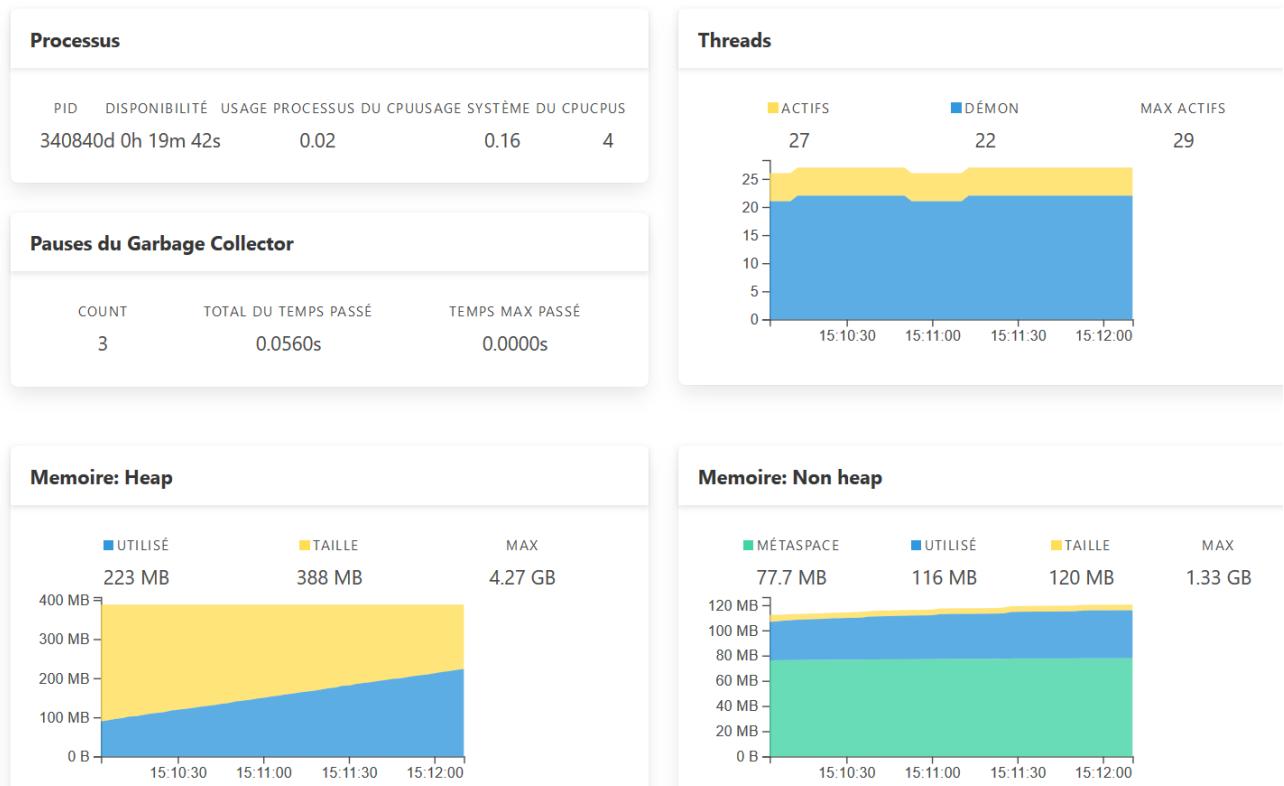
spring-boot-application
Id: e17564762c69

↖ <http://LAPTOP-DDC:8181/spring-boot-backend> ↗ <http://LAPTOP-DDC:8181/spring-boot-backend/actuator>
之心 <http://LAPTOP-DDC:8181/spring-boot-backend/actuator/health>

Info	État
app name: spring-boot-backend description: appli spring-boot , backend	Instance UP

Métadonnées

startup 2020-02-03T14:52:36.2960337+01:00



Code minimaliste de l'application "spring-admin"

pom.xml

```
.....
<properties>
    <java.version>1.8</java.version>
    <spring-boot-admin.version>2.2.1</spring-boot-admin.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>de.codecentric</groupId>
```

```

<artifactId>spring-boot-admin-starter-server</artifactId>
</dependency>
...
</dependencies>

<dependencyManagement>
<dependencies>
<dependency>
<groupId>de.codecentric</groupId>
<artifactId>spring-boot-admin-dependencies</artifactId>
<version>${spring-boot-admin.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
...
</project>
```

application.properties

```

server.servlet.context-path=/spring-admin
server.port=8787
logging.level.org=INFO

#this "spring-boot-admin server" app can monitor
#several "spring-boot-admin client" ordinary app with actuators

#spring-boot-admin SERVER properties:
spring.security.user.name=admin
spring.security.user.password=admin-pwd
```

SecurityConfig.java

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
```

```

SavedRequestAwareAuthenticationSuccessHandler successHandler
    = new SavedRequestAwareAuthenticationSuccessHandler();
successHandler.setTargetUrlParameter("redirectTo");
successHandler.setDefaultTargetUrl("/");

http.authorizeRequests()
    .antMatchers("/assets/**").permitAll()
    .antMatchers("/login").permitAll()
    .anyRequest().authenticated().and()
    .formLogin().loginPage("/login")
    .successHandler(successHandler).and()
    .logout().logoutUrl("/logout").and()
    .httpBasic().and()
    .csrf()
    .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
    .ignoringAntMatchers(
        "/instances",
        "/actuator/**"
    );
}
}

```

SpringAdminApplication.java

```

...
@SpringBootApplication
@EnableAdminServer //de.codecentric.boot.admin.server.config.EnableAdminServer
public class SpringAdminApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringAdminApplication.class, args);
        System.out.println("http://localhost:8787/spring-admin");
    }
}

```

.../...

Partie spring-boot-admin-starter-client à ajouter dans application ordinaire :

dans *pom.xml* :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>2.2.1</version>
</dependency>
```

dans *application.properties*

```
management.endpoints.web.exposure.include=*
#management.security.enabled=false or .antMatchers("/actuator/**").permitAll() or ...

#spring.boot.admin.client params to register in spring.boot.admin.server "spring-admin" :
spring.boot.admin.client.url=http://localhost:8787/spring-admin
spring.boot.admin.client.username=admin
spring.boot.admin.client.password=admin-pwd
```

+ tous les paramétrages "spring-actuator" habituels .

XVIII - Annexe – Web Services REST (concepts)

1. Deux grands types de WS (REST et SOAP)

2 grands types de services WEB: **SOAP/XML** et **REST/HTTP**

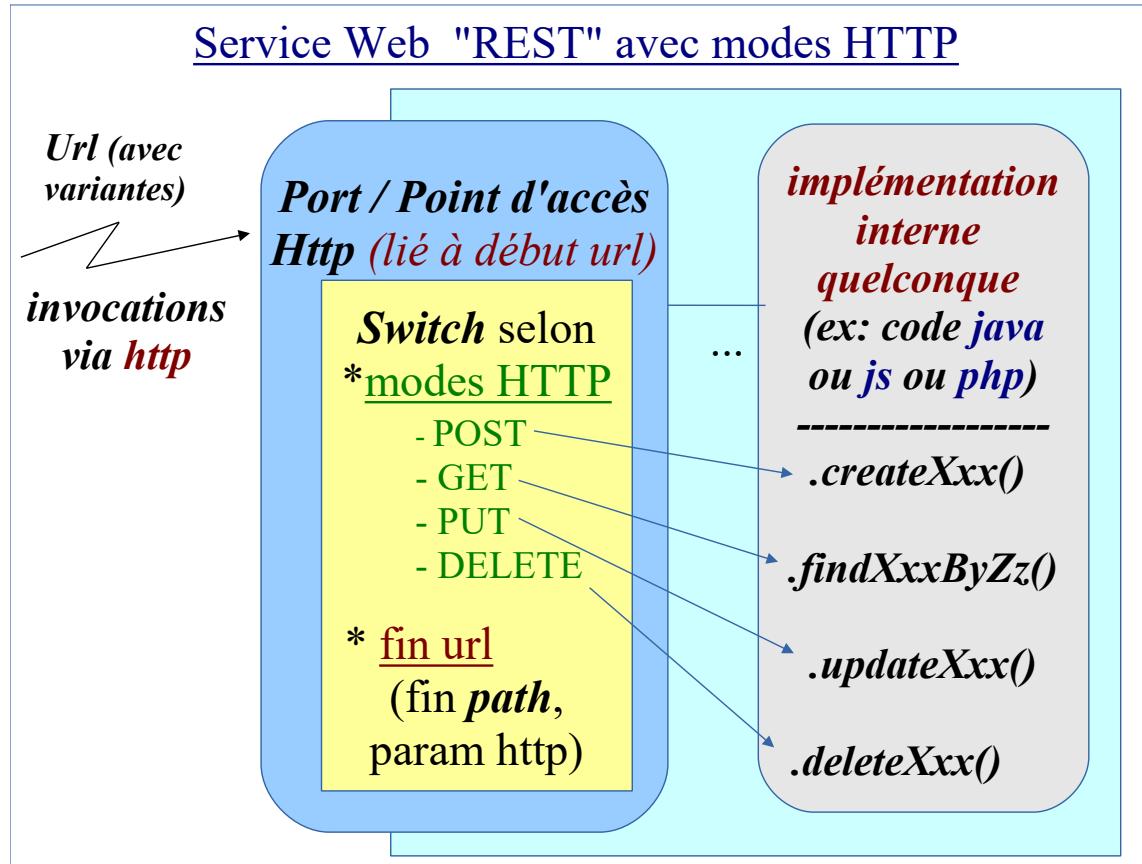
WS-* (SOAP / XML)

- "Payload" systématiquement en **XML** (*sauf pièces attachées / HTTP*)
- Enveloppe SOAP en XML (*header facultatif pour extensions*)
- Protocole de transport au choix (**HTTP, JMS, ...**)
- Sémantique quelconque (*appels méthodes*) , description **WSDL**
- Plutôt orienté Middleware SOA (**arrière plan**)

REST (HTTP)

- "Payload" au choix (**XML , HTML , JSON, ...**)
- Pas d'enveloppe imposée
- Protocole de transport = toujours **HTTP**.
- Sémantique "**CRUD**" (*modes http PUT,GET,POST,DELETE*)
- Plutôt orienté IHM Web/Web2 (**avant plan**)

1.1. Caractéristiques clefs des web-services "REST" / "HTTP"



Points clefs des Web services "REST"

Retournant des données dans un format quelconque ("XML", "JSON" et éventuellement "txt" ou "html") les web-services "REST" offrent des résultats qui nécessitent généralement peu de re-traitements pour être mis en forme au sein d'une IHM web.

Le format "**au cas par cas**" des données retournées par les services REST permet peu d'automatisme(s) sur les niveaux intermédiaires.

Souvent associés au format "**JSON**" les web-services "REST" conviennent parfaitement à des appels (ou implémentations) au sein du langage javascript .

La **relative simplicité des URLs d'invocation des services "REST"** permet des appels plus immédiats (*un simple href="..." suffit en mode GET pour les recherches de données*) .

La **compacité/simplicité des messages "JSON"** souvent associés à "REST" permet d'obtenir d'**assez bonnes performances** .

2. Web Services "R.E.S.T."

REST = style d'architecture (conventions)

REST est l'acronyme de **R**epresentational **S**tate **T**ransfert.

C'est un **style d'architecture** qui a été décrit par *Roy Thomas Fielding* dans sa thèse «*Architectural Styles and the Design of Network-based Software Architectures*».

L'information de base, dans une architecture REST, est appelée **ressource**.

Toute information (à sémantique stable) qui peut être nommée est une ressource: un article , une photo, une personne , un service ou n'importe quel concept.

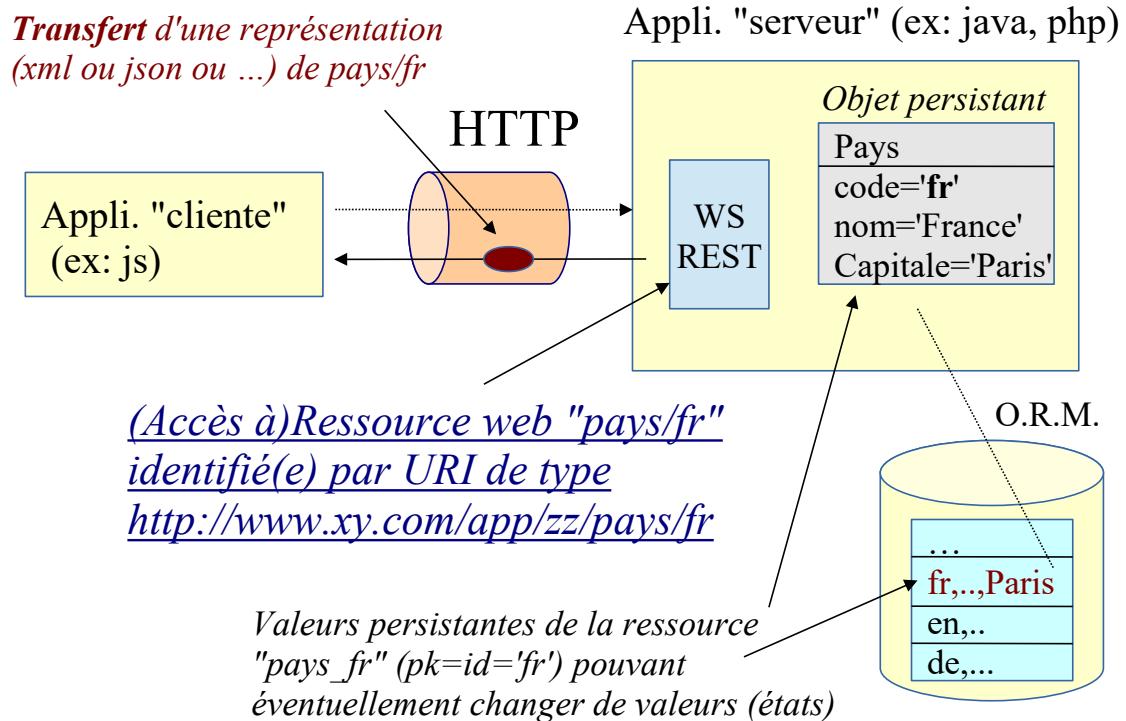
Une ressource est identifiée par un **identificateur de ressource**. Sur le web ces identificateurs sont les **URI** (Uniform Resource Identifier).

NB: dans la plupart des cas, une ressource REST correspond indirectement à un enregistrement en base (avec la *clef primaire* comme partie finale de l'uri "identifiant").

Les composants de l'architecture REST manipulent ces ressources en **transférant à travers le réseau** (via HTTP) des **représentations de ces ressources**.

Sur le web, on trouve aujourd'hui le plus souvent des représentations au format **HTML , XML ou JSON**.

REST : transferts de représentations de ressources



REST et principaux formats (xml,json)

Une invocation d'URL de service REST peut être accompagnée de données (en entrée ou en sortie) pouvant prendre des formats quelconques : **text/plain , text/html , application/xml , application/json , ...**

Dans le cas d'une lecture/recherche d'informations , le format du résultat retourné pourra (selon les cas) être :

- imposé (en dur) par le code du service REST .

- au choix (xml , json) et précisé par une partie de l'url

- au choix (xml , json) et précisé par le champ "Accept :" de l'entête HTTP de la requête. (exemple: Accept: application/json) .

Dans tous les cas, la réponse HTTP devra avoir son format précisé via le champ habituel **Content-Type: application/json** de l'entête.

Format JSON (JSON = *JavaScript Object Notation*)

Les 2 principales caractéristiques de JSON sont :

- Le principe de clé / valeur (map)
- L'organisation des données sous forme de tableau

```
[  
  {  
    "nom": "article a",  
    "prix": 3.05,  
    "disponible": false,  
    "descriptif": "article1"  
  },  
  {  
    "nom": "article b",  
    "prix": 13.05,  
    "disponible": true,  
    "descriptif": null  
  }]  
]
```

Les types de données valables sont :

- tableau
- objet
- chaîne de caractères
- valeur numérique (entier, double)
- booléen (true/false)
- null

une liste d'articles

une personne

```
{  
  "nom": "xxxx",  
  "prenom": "yyyy",  
  "age": 25  
}
```

REST et méthodes HTTP (verbes)

Les **méthodes HTTP** sont utilisées pour indiquer la **sémantique des actions demandées** :

- **GET** : lecture/recherche d'information
- **POST** : envoi d'information
- **PUT** : mise à jour d'information
- **DELETE** : suppression d'information

Par exemple, pour récupérer la liste des adhérents d'un club, on peut effectuer une requête de type **GET** vers la ressource **http://monsite.com/adherents**

Pour obtenir que les adhérents ayant plus de 20 ans, la requête devient
http://monsite.com/adherents?ageMinimum=20

Pour supprimer numéro 4, on peut employer une requête de type **DELETE** telle que **http://monsite.com/adherents/4**

Pour envoyer des informations, on utilise **POST** ou **PUT** en passant les informations dans le corps (invisible) du message HTTP avec comme URL celle de la ressource web que l'on veut créer ou mettre à jour.

Exemple concret de service REST : "Elevation API"

L'entreprise "**Google**" fourni gratuitement certains services WEB de type REST.

"Elevation API" est un service REST de Google qui renvoie l'altitude d'un point de la planète selon ses coordonnées (latitude ,longitude) .

La documentation complète se trouve au bout de l'URL suivante :

<https://developers.google.com/maps/documentation/elevation/?hl=fr>

Sachant que les coordonnées du Mont blanc sont :

Lat/Lon : 45.8325 N / 6.86417 E (GPS : 32T 334120 5077656)

Les invocations suivantes (du service web rest "api/elevation")

<http://maps.googleapis.com/maps/api/elevation/json?locations=45.8325,6.86417>

<http://maps.googleapis.com/maps/api/elevation/xml?locations=45.8325,6.86417>

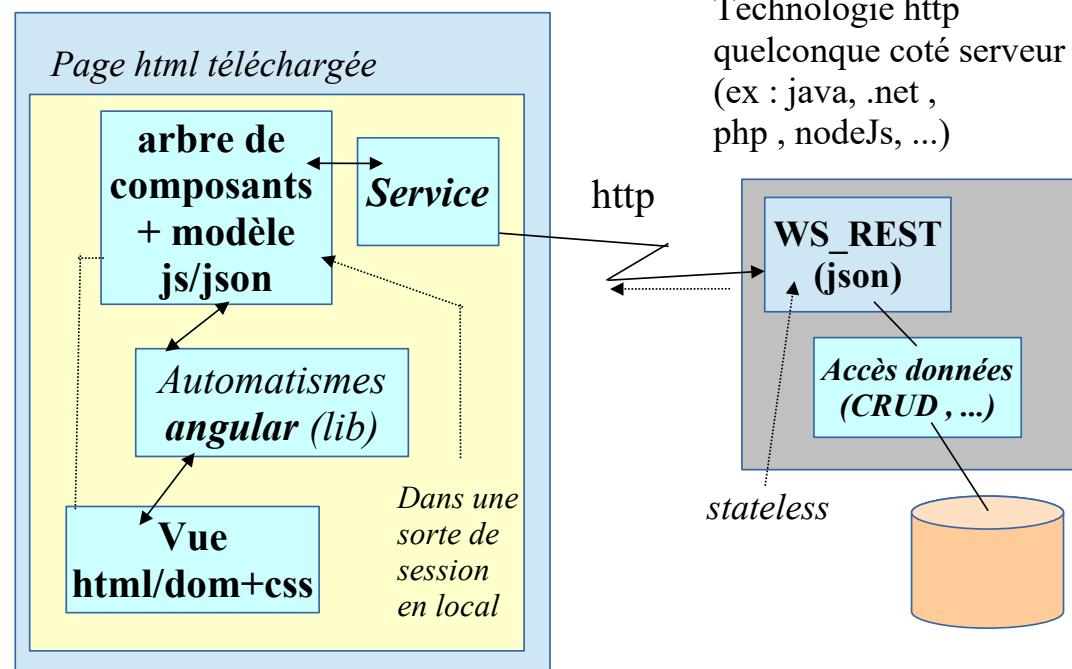
donne les résultats suivants "json" ou "xml":

```
{
  "results" : [
    {
      "elevation" : 4766.466796875,
      "location" : {
        "lat" : 45.8325,
        "lng" : 6.86417
      },
      "resolution" : 152.7032318115234
    },
    {
      "status" : "OK"
    }
  ]
}
```

```
?xml version="1.0" encoding="UTF-8"?
<ElevationResponse>
  <status>OK</status>
  <result>
    <location>
      <lat>45.8325000</lat>
      <lng>6.8641700</lng>
    </location>
    <elevation>4766.4667969</elevation>
    <resolution>152.7032318</resolution>
  </result>
</ElevationResponse>
```

Angular (positionnement)

Coté client (navigateur)



Conventions sur URL / Path des ressources REST

Type requêtes	HTTP Method	URL ressource(s) distante(s)	Request body	Réponse JSON
Recherche multiple	GET	.../product .../product?crit1=v1&crit2=v2	vide	Liste/tableau d'objets
Recherche par id	GET	.../product/idRes (avec idRes=1,...)	vide	Objet JSON
Ajout (seul)	POST	.../product	Objet JSON	Objet JSON avec id quelquefois calculé (incr)
Mise à jour (seule)	PUT	.../product/idRes ou .../product	Objet JSON avec .id	Objet JSON mis à jour
SaveOr Update	POST	.../product	Objet JSON	Objet JSON ajouté (auto incr id) ou modifié
suppression	DELETE	.../product/idRes	vide	Statut et message
Autres/product-action/opXy/...

2.1. Statuts HTTP (code d'erreur ou ...)

Catégories de code/statut HTTP :

1xx	Information (rare)
2xx (ex : 200)	Succès
3xx	Redirection
4xx	Erreur du client
5xx (ex : 500)	Erreur du serveur

Principaux codes/statuts en cas de succès ou de redirection:

200 , OK	Requête traitée avec succès. La réponse selon méthode de requête utilisée
201 , Created	Requête traitée avec succès et création d'un document.
204 , No Content	Requête traitée avec succès mais pas d'information à renvoyer.
301 , Moved Permanently	Document déplacé de façon permanente
304 , Not Modified	Document non modifié depuis la dernière requête

Principaux codes d'erreurs :

400 , Bad Request	La syntaxe de la requête est erronée (ex : invalid argument)
401 , Unauthorized	Une authentification est nécessaire pour accéder à la ressource.
403 , Forbidden	authentification effectuée mais manque de droits d'accès (selon rôles, ...)
404 , Not Found	Ressource non trouvée.
409 , Conflict	La requête ne peut être traitée en l'état actuel.
...	<i>liste complète sur</i> Liste_des_codes_HTTP">https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP
500 , Internal Server Error	Erreur interne (vague) du serveur (ex ; bug , exception , ...).
501, Not Implemented	Fonctionnalité réclamée non supportée par le serveur
503 , Service Unavailable	Service temporairement indisponible ou en maintenance.

2.2. Variantes classiques

Réponses plus ou moins détaillées (simple "http status" ou bien "message json")

Lorsqu'un serveur répond à une requête en mode POST , il peut soit :

- retourner le "https status" **201/CREATED** et une réponse JSON comportant toute l'entité sauvegardée coté serveur avec souvent l'*id* (clef primaire) automatiquement généré ou incrémenté
`{ "id" : "a345b6788c335d56" , "name" : "toto" , ... }`

- se contenter de renvoyer le "http status" **201/CREATED** avec aucun message de réponse mais avec le champ **Location:** `/type_entite/idxy` comportant au moins l'*id* de la resource enregistrée au sein de l'entête HTTP de la réponse .

L'application cliente pourra alors effectuer un second appel en mode GET avec une fin d'URL en `/type_entite/idxy` si elle souhaite récupérer tous les détails de l'entité sauvegardée .

- combiner les 2 styles de réponses (champ Location ET réponse JSON)

Lorsqu'un serveur répond à une requête en mode DELETE , il peut soit :

- se contenter de renvoyer le "http status" **204/NO_CONTENT** et aucun message
- retourner le "https status" **200/OK** et une réponse JSON de type
`{ "message" : "resource of id ... successfully deleted" }`

Lorsqu'un serveur répond à une requête en mode PUT , il peut soit :

- se contenter de renvoyer le "http status" **204/NO_CONTENT** et aucun message
- retourner le "https status" **200/OK** et une réponse JSON comportant toutes les valeurs de l'entité mise à jour du coté serveur , exemple:
`{ "id" : "a345b6788c335d56" , "name" : "titi" , ... }`

On peut éventuellement envisager que le serveur réponde **par défaut** aux modes PUT et DELETE par un simple **204/NO_CONTENT** et qu'il réponde par **200/OK + un message JSON** si le paramètre http optionnel `?v=true` ou `?verbose=true` est présent en fin de l'URL de la requête .

Identifiant de la resource à modifier en mode PUT placé en fin d'URL ou bien dans le corps de la requête HTTP, ou bien les deux.

Lorsqu'un serveur reçoit une requête de mise à jour en mode PUT , l'*id* de l'entity peut soit être précisée en fin d'URL , soit être précisée dans les données json de la partie body et si l'information est renseignée des 2 façons elle ne doit pas être incohérente .

Le serveur peut éventuellement faire l'effort de récupérer l'*id* de l'une ou des deux façons envisageables et peut renvoyer **400/BAD_REQUEST** si l'*id* de l'entité à mettre à jour n'est pas renseigné ou bien incohérent.

2.3. Safe and idempotent REST API

Une Api "Rest" désigne un ensemble de Web-services liés à un certain domaine fonctionnel (ex : gestion des stocks ou facturation ou ...)

Un appel "HTTP" vers une api-rest est dit "*safe*" s'il n'engendre pas de modifications du coté des ressources du serveur ("*safe*" = "*readonly*") .

En mathématique , une fonction est dite "idempotente" si plusieurs appels successifs avec les mêmes paramètres retournent toujours le même résultat.

Au niveau d'une api-rest , une invocation HTTP (ex : *GET* , *PUT* ou *DELETE*) est dite "idempotente" si plusieurs appels successifs avec les mêmes paramètres engendrent un même état résultat" au niveau du serveur .

Mais la réponse HTTP peut cependant varier .

Exemple : premier appel à "delete xyz/567" --> return "200/OK" ou "204/NO_CONTENT"

et second appel à "delete xyz/567"--> return 404 / notFound

mais dans les 2 cas , la ressource de type "xyz" et d'id=567 est censée ne plus exister .

Le DELETE est donc généralement considéré comme idempotent .

	safe	idempotent
GET (et HEAD,OPTIONS)	y	y
PUT	n	y
DELETE	n	y
POST	n	n

Intérêt de l'impotence comportementale du coté serveur :

Une application cliente doit souvent passer par des intermédiaires pour véhiculer une requête HTTP jusqu'au serveur . Certains mécanismes intermédiaires considèrent "internet / http" comme pas fiable à 100 % et vont quelquefois effectuer plusieurs retransmissions d'une requête si la première tentative échoue . il vaut mieux donc que le serveur se comporte de manière idempotente dans un maximum de cas .

Bien que le vocabulaire "idempotency" ne soit pas du tout approprié , il est tout de même conseillé de retourner des réponses HTTP dans un format assez homogène vers le client pour que celui-ci soit simple à programmer (pas trop de if ... else ...)

Dans tous les cas , bien documenter "comportements & réponses" d'une apit rest .

3. Test de W.S. REST via Postman

L'application "postman" (téléchargeable depuis l'url <https://www.postman.com/downloads/>) existe depuis longtemps et est souvent considérée comme l'application de référence pour tester les web services "REST".

NB1 : après le premier lancement , il n'est pas obligatoire de s'enregistrer (créer un compte) pour utiliser l'application , on peut cliquer sur un lien à peine visible plus bas que la boite de dialogue nous invitant à nous enregistrer et l'on peut d'une manière générale fermer toutes les "popups" et créer un nouvel onglet de requête pour paramétrier et lancer un test.

NB2 : A une certaine époque , "postman" pouvait s'utiliser en tant que plugin pour le navigateur "chrome" . Ce plugin est maintenant "deprecated" (plus maintenu) .

3.1. paramétrages "postman" pour une requête en mode "post"

The screenshot shows the Postman interface with the following configuration:

- Request URL:** http://localhost:8080/serverSoap/ws/rest/devise
- Method:** POST
- Headers (1):**
 - Content-Type: application/json
- Body:**
 - Raw JSON payload:

```

1 <?
2 "codeDevise" : "MS2",
3 "tauxChange" : 1.4568
4 ?>

```
- Response Status:** 200 OK

3.2. Exemple de réponses précises reçues et affichées par "postman"

POST , http://localhost:8282/login-api/public/auth , Content-Type : application/json ,

request body : { "username": "admin1" , "password" : "pwdadmin1" , "roles" : "admin,user" }

==> 200 ok

et responseBody :

```
{
  "username": "admin1",
  "status": true,
  "message": "successful login",
  "token":
    "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiYWRtaW4xIiwicm9sZXMiOiJh
    ZG1pbix1c2VyIiwiaWF0IjoxNTk2Nzk2MTI0LCJleHAiOjE1OTY4MDMzMjQsImlzcyI6Imh0dH
    A6Ly93d3cubXljb21wYW55In0.wBQHJPN20VE7tzrF8vk3Cq9FltiQQOf2RETDMSB19ho",
  "roles": "admin,user"
}
```

Si requête comportant { "username": "admin1" , "password" : "abcdef" , "roles" : "admin,user" }

alors réponse de ce type :

```
{
  "username": "admin1",
  "status": false,
  "message": "login failed (wrong password)",
  "token": null,
  "roles": null
}
```

DELETE , http://localhost:8282/devise-api/private/role_admin/devise/m3875

==> 200 ok

et responseBody :

```
{
  "action": "devise with code=m3875 was deleted"
}
```

ou bien (suite à un second appel successif) :

==> 404 not found

```
{
  "errorCode": "404",
  "message": "deleteOne exception with id=m3875"
}
```

4. Test de W.S. REST via curl

curl (*command line url*) est un programme utilitaire (d'origine linux) permettant de déclencher des requêtes HTTP via une simple ligne de commande.

Via certaines options , curl peut effectuer des appels en mode "GET" , "POST" , "DELETE" ou "PUT".

Ceci peut être très pratique pour tester rapidement un web service REST via quelques lignes de commandes placées dans un script réutilisable (.bat, .sh ,) .

lancer_curl.bat

```
cd /d "%~dp0"
REM instructions qui vont bien
set URL=http://localhost:8081/my-api/info/1
curl %URL%
pause
```

curl fonctionne en **mode GET par défaut** si pas de -d (*pas de data*)

```
curl %URL%
REM "verbose" (-v) très pratique pour connaître les détails de la communication réseau
curl -v %URL%
```

```
curl -o out.json %URL%
```

pour stocker la réponse dans un fichier texte (ici out.json)

curl fonctionne en mode **POST** par défaut avec data (**-d** ...)

curl fonctionne en mode PUT si **-X PUT** et mode DELETE si **-X DELETE**

appel au format par défaut (`application/x-www-form-urlencoded`)

si pas d'option -H "Content-Type:" au niveau de la requête
alors par défaut logique champ/paramètre de formulaire en mode POST avec

-d paramName1=valeur1 -d paramName2=valeur2 ...

Exemple :

set URL=clientIdPassword:secret:@localhost:8081/basic-oauth-server/oauth/token

set PWD=d8dfc382-e012-491a-8d03-ca6ad9d81083

curl %URL% -d grant_type=password -d username=user -d password=%PWD%

Requête au format "application/json" :

NB : en version windows , curl ne gère pas bien les simples quotes et il faut préfixer les " internes par des \

```
curl %LOGIN_URL% -H "Content-Type: application/json"  
-d "{\"username\":\"member1\", \"password\": \"pwd1\"}"
```

il vaut mieux donc utiliser un fichier pour les données en entrée :

```
curl %LOGIN_URL% -H "Content-Type: application/json" -d @member1-login-request.json
```

avec

member1-login-request.json

```
{  
"username": "member1",  
"password": "pwd1"  
}
```

Authentification avec curl :

```
curl --user myUsername:myPassword ... permet une "BASIC HTTP AUTHENTICATION"
```

ou bien

```
curl -H "Authorization: Bearer b1094abc.._ou_.autre_.jeton" permet une demande d'autorisation  
en mode "Bearer / au porteur de jeton" (jeton à préalablement récupérer via login ou autre )
```

5. Api Key

Un web service hébergé par une entreprise et rendu accessible sur internet a un certain coût de fonctionnement (courant électrique , serveurs ,) .

Pour limiter des abus (ex : appel en boucle) ou bien pour obtenir un paiement en contre partie d'une bonne qualité de service , un web service public est souvent invocable que si l'on renseigne une "api_key" (au niveau de l'URL ou bien au niveau de l'entête la requête HTTP).

Une "api_key" est très souvent de type "**uuid/guid**" .

Critères d'une api key :

- lié à un abonnement (gratuit ou payant) , ex : compte utilisateur / compte d'entreprise
- ne doit idéalement pas être diffusé (à garder secret)
- souvent lié à un compteur d'invocations (limite selon prix d'abonnement)
- doit pouvoir être administré (régénéré si perdu/volé , ...) et les modifications doivent pouvoir être immédiatement ou rapidement prises en compte.

Exemple :

Le site <https://fixer.io> héberge un web service REST permettant de récupérer les taux de change (valeurs de "USD" , "GBP" , "JPY" , ... vis à vis de "EUR" par défaut).

Début 2018, ce web service était directement invocable sans "api_key" .

Courant 2018, ce web service est maintenant invocable qu'avec une "api_key" liée à un compte utilisateur "gratuit" ou bien "payant" selon le mode d'abonnement (options, fréquence d'invocation,).

URL d'appel sans "api_key" : <http://data.fixer.io/api/latest>

Réponse :

```
{
  "success":false,
  "error":{ "code":101, "type":"missing_access_key",
    "info":"You have not supplied an API Access Key. [Required format:
      access_key=YOUR_ACCESS_KEY]"
  }
}
```

URL d'invocation avec api_key valide :

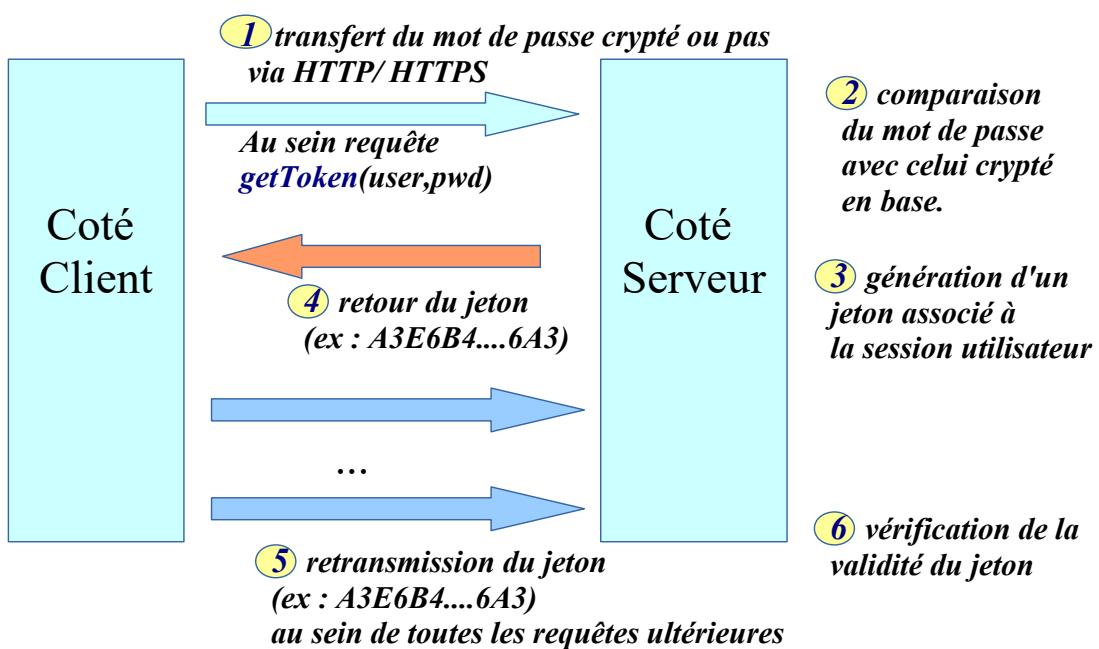
http://data.fixer.io/api/latest?access_key=26ca93ee7.....aaa27cab235

```
{
"success":true, "timestamp":1538984646, "base":"EUR", "date":"2018-10-08",
"rates":
{"AED":4.224369,...,"DKK":7.460075,"DOP":57.311592,"DZD":136.091172,"EGP":20.596249,
"ERN":17.250477,"ETB":31.695652,"EUR":1,"FJD":2.46956,"FKP":0.88584,"GBP":0.879667,
...,"JPY":130.858498,...,"USD":1.15005,...,"ZWL":370.724343}
}
```

6. Token d'authentification

6.1. Tokens : notions et principes

Jeton ("token") d'authentification valide le temps d'une session utilisateur



Plusieurs sortes de jetons/tokens

Il existe plusieurs sortes de jetons (normalisés ou pas).

Dans le cas le plus simple, un **jeton est généré aléatoirement** (ex : **uuid** ou ...) et sa **validation consiste essentiellement à vérifier son existence** en tentant de le récupérer quelque part (*en mémoire ou en base*) et éventuellement à vérifier une date et heure d'expiration.

JWT (Json Web Token) est un **format particulier de jeton** qui **comporte 3 parties** (une entête technique , un paquet d'informations en clair (ex : username , email , expiration, ...) au format JSON et une signature qui ne peut être vérifiée qu'avec la clef secrète de l'émetteur du jeton.

6.2. Bearer Token (au porteur) / normalisé HTTP

Bearer token (jeton au porteur) et transmission

Le champ **Authorization**: normalisé d'une entête d'une requête HTTP peut comporter une valeur de type **Basic** ... ou bien **Bearer** ...

Le terme anglais "**Bearer**" signifiant "**au porteur**" en français indique que la simple possession d'un jeton valide par une application cliente devrait normalement , après transmission HTTP, permettre au serveur d'autoriser le traitement d'une requête (après vérification de l'existence du jeton véhiculé parmi l'ensemble de ceux préalablement générés et pas encore expirés).

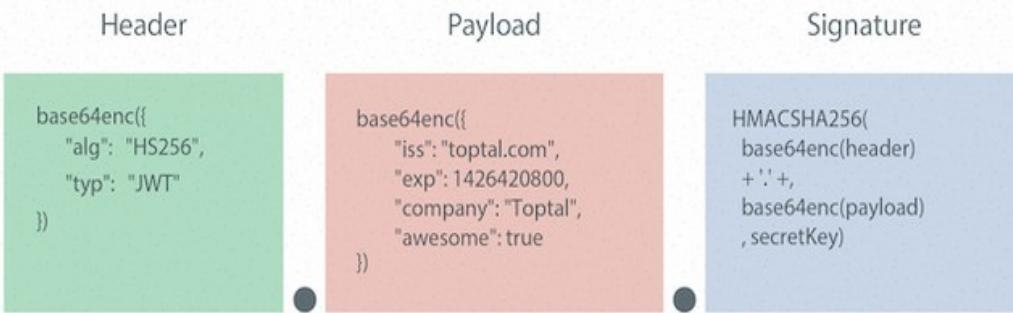
NB: Les "bearer token" sont utilisés par le protocole "O2Auth" mais peuvent également être utilisés de façon simple sans "O2Auth" dans le cadre d'une authentification "sans tierce partie" pour API REST.

NB2 : un "bearer token" peut éventuellement être au format "JWT" mais ne l'est pas toujours (voir rarement) en fonction du contexte.

6.3. JWT (Json Web Token)



Structure jeton "JWT / Json Web Token"



Exemple:

`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJ0b3B0YWwuY29tIiwiZXhwIjoxNDI2NDIwODAwLCJodHRwOi8vdG9wdGFsLmNvbS9qd3RfY2xhaW1zL2lzX2FkbWluIjp0cnVILCJjb21wYW55JjoiVG9wdGFsIiwiYXdlc29tZSI6dHJ1ZX0.yRQYnWzskCZUxPwaQupWkiUzKELZ49eM7oWxAQK_ZXw`

NB : "iss" signifie "issuer" (émetteur) , "iat" : issue at time
 "exp" correspond à "date/heure expiration" . Le reste du "payload" est libre (au cas par cas) (ex : "company" et/ou "email" , ...)

XIX - Annexe – WebSockets , STOMP (Spring)

1. WebSockets HTML5 (standard)

Les "WebSockets" constituent une adaptation "HTTP" des classiques sockets "tcp/ip" .

C'est une sorte d'annexe/extension vis à vis du protocole HTTP .

Ayant leurs propres préfixes (schemas : ws : , wss :) , les "WebSockets" peuvent également être vues comme un nouveau protocole (avec même la notion de sous protocole possible tel que STOMP)

En tant que "sockets" , une "WebSockets" est un **canal de communication bi-directionnel établi durablement (tant que pas fermé)** entre un client et un serveur par exemple .

Ce canal bi-directionnel peut servir à spontanément envoyer (dans les 2 sens) des messages dans format quelconque (texte , json, ... ou binaire).

1.1. Fonctionnalités des WebSockets

Les "webSockets" sont surtout utilisés dans une logique de "**push**" ou "**subscribe/publish**" .

Une fois une connexion établie , le serveur peut spontanément envoyer de nouvelles valeurs (qui viennent de changer) vers le coté client/navigateur sans que celui-ci soit obligé d'effectuer une requête préalable d'actualisation .

En règle générale, de nombreux clients sont simultanément connectés à un même serveur .

Le serveur peut alors via une simple boucle diffuser une information vers tous les clients connectés via une websocket active .

Application classique :

- discussion en tant réel : "chat" , messagerie instantanée
- actualisation automatique de graphique (ex : SVG, canvas, ...) dès qu'une valeur change
- tableau (board) partagé en équipe
- toute autre communication en mode push (avec ou sans RxJs / mode réactif) .

1.2. Principe de fonctionnement

Une connexion "WebSocket" s'effectue en partant d'une connexion "http" existante puis en "upgradant" celle-ci durant une phase d'échange d'informations appelée "**handshake** " .

Le client envoi une requête HTTP de ce type

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGh1IHNhbXBsZSBub25jZQ==
Sec-WebSocket-Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 6
```

Le serveur doit s'il accepte l'upgrade , renvoyer une réponse de ce type :

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Au niveau du paramètre du constructeur `WebSocket()` de l'api `html5/javascript`, l'url d'une `websocket` à préciser pour établir une connexion ne commence pas par `http://` ni par `https://` mais par `ws://` ou `wss://` . Cependant , comme la connexion `ws://` est un upgrade d'une connexion HTTP , le numéro de port utilisé par les "websockets" reste le standard **80** et donc pas de soucis en général pour que les requêtes/réponses puissent passer à travers proxy ou firewall .

Une fois la connexion établie , Chaque protagoniste (client et serveur) voit l'autre coté du canal de communication comme un "**endpoint**" vers lequel on peut spontanément envoyer des messages via une méthode `.send()` .

Du coté réception , les méthodes "callback" suivantes seront automatiquement appelées :

- **onopen** : ouverture d'une WebSocket
- **onmessage** : réception d'un message
- **onerror** : erreur(s) survenue(s)
- **onclose** : fermeture de WebSocket (de l'autre coté)

1.3. Exemple de "chat"(WebSocket) : code client "html5+javascript"

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>  <title>Apache Tomcat WebSocket Examples: Chat</title>
<style type="text/css">...</style>
<script type="application/javascript"><![CDATA[
    "use strict";
    var Chat = {};//custom literal js object

    Chat.socket = null;
    Chat.connect = (function(host) {
        if ('WebSocket' in window) {
            Chat.socket = new WebSocket(host);
        } else if ('MozWebSocket' in window) {
            Chat.socket = new MozWebSocket(host);
        } else {
            Console.log('Error: WebSocket is not supported by this browser.');
            return;
        }

        Chat.socket.onopen = function () {
            Console.log('Info: WebSocket connection opened.');
            document.getElementById('chat').onkeydown = function(event) {
                if (event.keyCode == 13) {
```

```

        Chat.sendMessage();
    }
};

Chat.socket.onclose = function () {
    document.getElementById('chat').onkeydown = null;
    Console.log('Info: WebSocket closed.');
};

Chat.socket.onmessage = function (message) {
    Console.log(message.data);
};

Chat.initialize = function() {
    if (window.location.protocol == 'http:') {
        Chat.connect('ws://' + window.location.host + '/examples/websocket/chat');
    } else {
        Chat.connect('wss://' + window.location.host + '/examples/websocket/chat');
    }
};

Chat.sendMessage = (function() {
    var message = document.getElementById('chat').value;
    if (message != "") {
        Chat.socket.send(message);
        document.getElementById('chat').value = "";
    }
});

var Console = {};//custom literal js object

Console.log = (function(message) {
    var console = document.getElementById('console');
    var p = document.createElement('p');
    p.style.wordWrap = 'break-word';
    p.innerHTML = message;
    console.appendChild(p);
    while (console.childNodes.length > 25) {
        console.removeChild(console.firstChild);
    }
    console.scrollTop = console.scrollHeight;
});

Chat.initialize();
]]></script>
</head>
<body>
<p>
    <input type="text" placeholder="type and press enter to chat" id="chat" />
</p>
<div id="console-container">
    <div id="console"/>
</div>
</div></body></html>

```

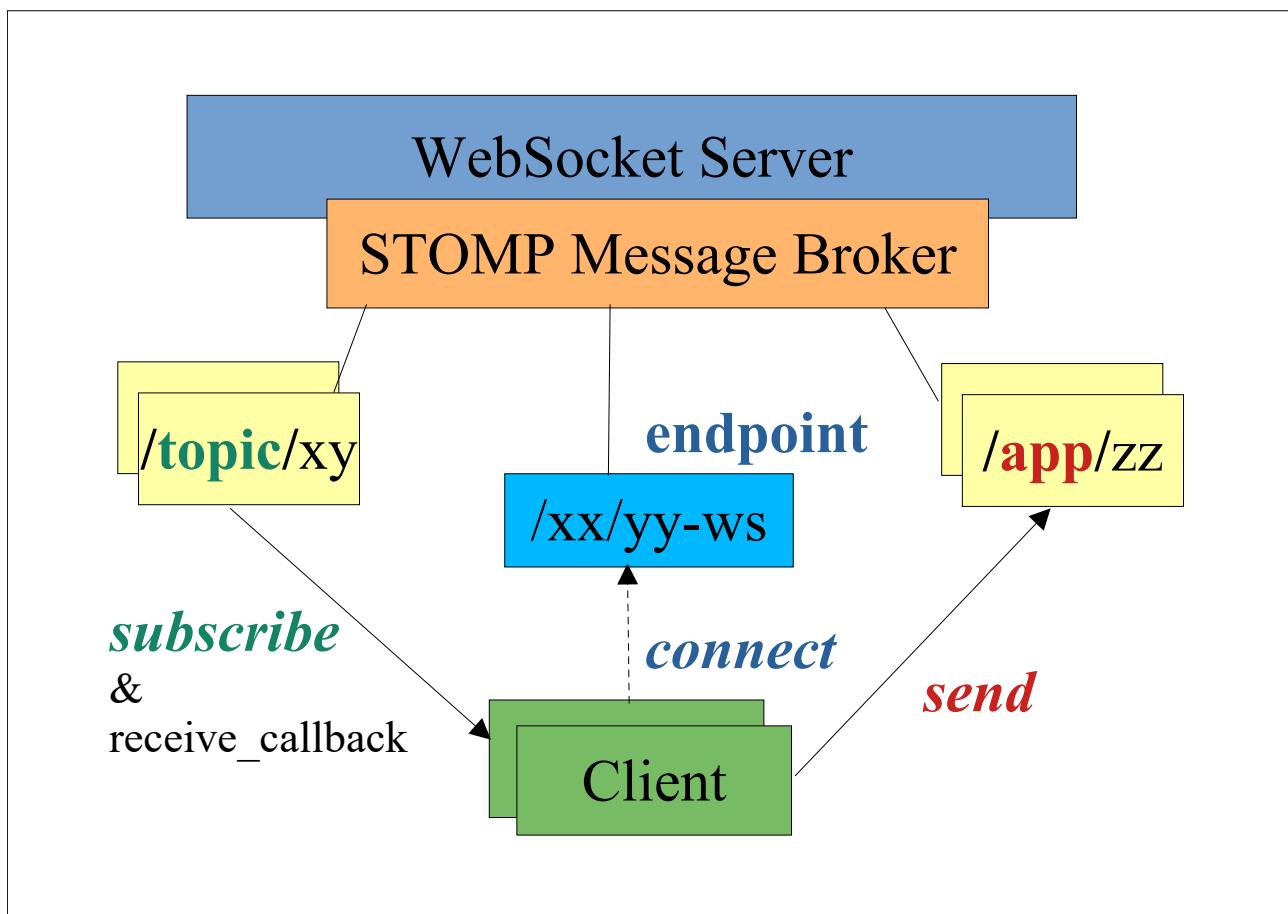
2. WebSockets avec Spring et STOMP

2.1. STOMP

STOMP (Streaming Text Oriented Messaging Protocol): (protocole textuel orienté messages) est un protocole de communication, une branche du WebSocket.

Lorsque le client et le server se contactent via ce protocole, ils enverront des données textuelles orientées messages.

La relation entre STOMP et WebSocket est un peu similaire à celle de HTTP et TCP.



2.2. Configuration

pom.xml partiel (en version Spring-boot)

```

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

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

```

```
</dependency> <!-- et indirectement spring-messaging et spring-websocket -->
```

WebSocketconfig.java

```
package tp.springwebsocket;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;
import org.springframework.web.socket.config.annotation.WebSocketMessageBrokerConfigurer;

@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/chat-ws").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.setApplicationDestinationPrefixes("/app");
        registry.enableSimpleBroker("/topic");
    }
}
```

```
public class ChatMessage {

    public enum MessageType { CHAT, JOIN, LEAVE }

    private String type; //MessageType as String
    private String sender;//or from
    private String content;//or text
    private String time;

    //+ get/set, default constructor , ...
}
```

@SpringBootApplication

```
public class SpringWebSocketApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringWebSocketApplication.class, args);
        System.out.println("http://localhost:8484/springWebSocket");
    }
}
```

src/main/resources/**application.properties**

```
server.servlet.context-path=/springWebSocket
server.port=8484
```

WebSocketController.java (*retransmission des messages*)

```
...
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.stereotype.Controller;

@Controller
public class WebSocketController {

    @MessageMapping("/chat") //input message received from /chat
    @SendTo("/topic/messages") //output message publish to /topic/messages
    public ChatMessage send_forward(ChatMessage message) throws Exception {
        String time = new SimpleDateFormat("HH:mm:ss").format(new Date());
        message.setTime(time);
        return message;
    }
}
```

src/main/resources/static/chat.html

```
<html>
<head>
    <title>Chat WebSocket</title>
    <script src=".js/sockjs.min.js"></script>
    <script src=".js/stomp.min.js"></script>
    <script src=".js/my-chat.js"></script>

</head>
<body onload="disconnect()">
    <div>
        <div>
            sender/from:<input type="text" id="from" placeholder="Choose a nickname"/> <br />
            <button id="connect" onclick="connect();">Connect</button>
            <button id="disconnect" disabled="disabled"
                   onclick="disconnect();">Disconnect</button>
        </div>
        <hr />
        <div id="conversationDiv">
            message:<input type="text" id="text" placeholder="Write a message..."/>
            <button id="sendMessage" onclick="sendMessage();">Send</button>
            <ul id="response"></ul>
        </div>
    </div>
</body>
</html>
```

src/main/resources/static/js/my-chat.js

```

var stompClient = null;

function connect() {
    let from = document.getElementById('from').value;
    if(from=="") return;
    let socket = new SockJS('/springWebSocket/chat-ws');
    stompClient = Stomp.over(socket);
    stompClient.connect({}, function(frame) {
        setConnected(true); console.log('Connected: ' + frame);

        stompClient.subscribe('/topic/messages', function(messageOutput) {
            showMessageOutput(JSON.parse(messageOutput.body));
        });

        stompClient.send("/app/chat", {},
                        JSON.stringify({ type: "JOIN" , sender: from, content: "is connected" }));
    });
}

function setConnected(connected) {
    document.getElementById('connect').disabled = connected;
    document.getElementById('disconnect').disabled = !connected;
    document.getElementById('conversationDiv').style.visibility
        = connected ? 'visible' : 'hidden';
    document.getElementById('response').innerHTML = "";
}

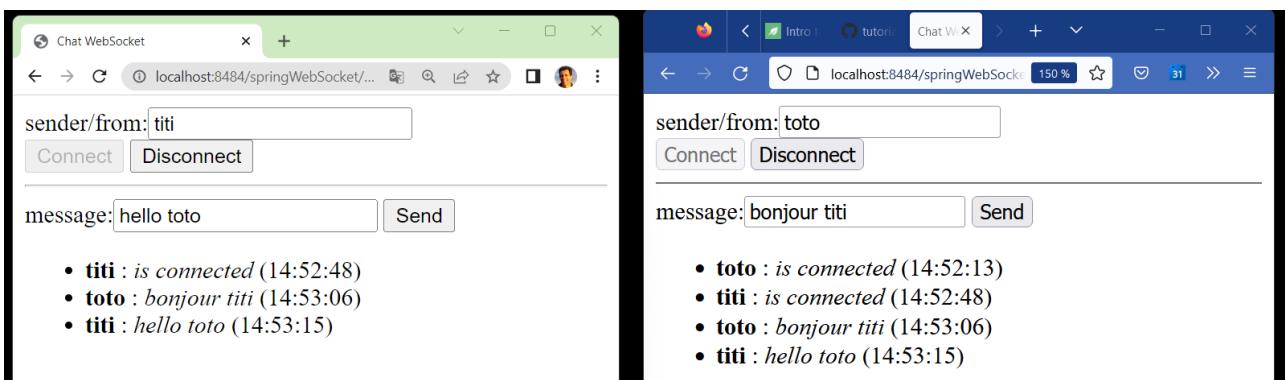
function disconnect() {
    let from = document.getElementById('from').value;
    if(stompClient != null) {
        stompClient.send("/app/chat", {},
                        JSON.stringify({ type: "LEAVE" , sender: from, content: "is disconnected" }));
        stompClient.disconnect();
    }
    setConnected(false);
    console.log("Disconnected");
}

function sendMessage() {
    let from = document.getElementById('from').value;
    let text = document.getElementById('text').value;
    stompClient.send("/app/chat", {},
                    JSON.stringify({ type: "CHAT" , sender: from, content: text /* , time : null */}));
}

function showMessageOutput(messageOutput) {
    var response = document.getElementById('response');
    var li = document.createElement('li');
    li.innerHTML=<b>"+messageOutput.sender + "</b> :<i> "
        + messageOutput.content + "</i> (" + messageOutput.time + ")";
}

```

```
response.appendChild(li);
```



Autre exemple plus élaboré avec *HandshakeInterceptor* et *WebSocketEventListener* :

<https://devstory.net/10719/creer-une-application-chat-simple-avec-spring-boot-et-websocket>

XX - Annexe – Spring Native Image

XXI - GraalVM et Graal Native image

1.1. GraalVM

GraalVM est une Machine Virtuelle (VM), Open Source, issue d'un projet de recherche commencé il y a plus de 10 ans chez Oracle Labs (anciennement Sun Labs) .

Cette nouvelle VM est maintenue par une communauté d'acteurs majeurs du net (Oracle, Amazon, Twitter, RedHat notamment avec Quarkus, VMWare pour l'intégration de son framework Spring, ...).

C'est une **nouvelle génération** de VM, **polyglotte**, c'est à dire qu'elle supporte de nombreux langages, même ceux qui ne génèrent pas de bytecode. A terme, elle pourrait remplacer l'actuelle VM HotSpot.

Techniquement :

- La VM **GraalVM** est couplée à un nouveau compilateur, **Graal**, écrit entièrement en Java (ce qui permet une compilation cyclique) :
- Il vise à remplacer le compilateur C2 utilisé pour le **JIT** de la VM **HotSpot** et qui est arrivé en fin de vie car trop complexe à faire évoluer (mélange d'assembleur, C, Java)
- Le compilateur Graal peut aussi faire de la compilation **AOT** (Ahead-Of-Time, à l'avance) aussi appelée compilation anticipée.

1.2. GraalVM Native Image

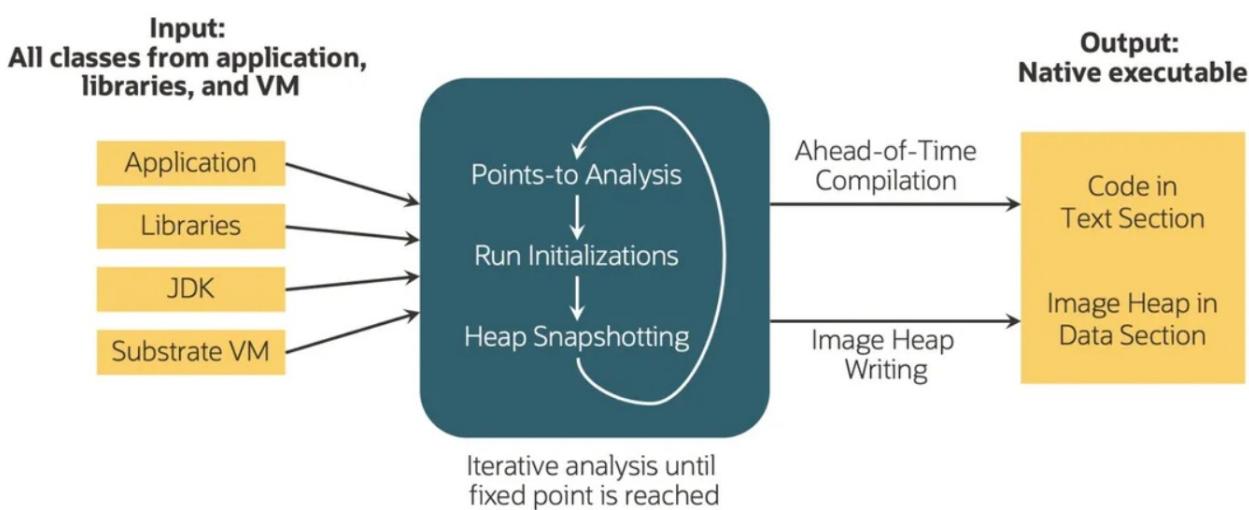
- **GraalVM Native Image** est une technologie de compilation ahead-of-time qui **génère des exécutables natifs**.
- **Les exécutables natifs sont idéaux pour les conteneurs et les déploiements cloud car ils sont petits, démarrent très rapidement et nécessitent beaucoup moins de CPU et de mémoire.**
- GraalVM Native Image bénéficie d'une adoption importante avec le support des principaux frameworks Java tels que Spring Boot, Micronaut, Quarkus, Gluon Substrate, etc.

Dans les grandes lignes , avec une JVM classique , la compilation et l'exécution se produisent en même temps (lors du démarrage) avec une longue période de chauffe .

Avec GraalVM Native Image, en mode AOT, le compilateur effectue toutes les compilations (jusqu'au stade binaire) pendant la construction et avant l'execution .

L'utilitaire GraalVM 'native-image' prend le bytecode Java en entrée et produit un exécutable natif.

Pour ce faire, l'utilitaire effectue une analyse statique du bytecode sous une hypothèse de monde fermé. Lors de l'analyse, l'utilitaire recherche tout le code que votre application utilise réellement et élimine tout ce qui est inutile(un peu comme un bundle javascript généré par webpack et angular).



Pour approfondir le sujet :

<https://scalastic.io/graalvm-microservices-java/>

<https://www.infoq.com/fr/articles/native-java-graalvm/> (source de l'image ci-dessus)

1.3. Spring Native

En 2022, le framework spring est en train d'intégrer "GraalVM Native Image" au sein des dernières versions Spring5/SpringBoot2 et des nouvelles versions Spring6/SpringBoot3 .

Article :

<https://www.infoq.com/fr/articles/native-java-spring-boot/>

<https://docs.spring.io/spring-native/docs/current/reference/htmlsingle/>

1.4. Quarkus (de RedHat)

<https://quarkus.io/>

<https://www.redhat.com/fr/topics/cloud-native-apps/what-is-quarkus>

<https://www.ionos.fr/digitalguide/serveur/configuration/quest-ce-que-quarkus/>

XXII - Spring Native Image

Attention:

- La construction d'une image native nécessite absolument **docker** et s'effectue en général sur une machine linux.
- La construction d'une image native prend beaucoup de temps (plusieurs minutes)
- C'est à considérer comme une **fonctionnalité encore un peu expérimentale** (avec pas mal de limitations)

Dans **pom.xml**

```
...
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.1</version>
</parent>
...
<plugin>
    <groupId>org.graalvm.buildtools</groupId>
    <artifactId>native-maven-plugin</artifactId>
</plugin>
...
...
```

En interne spring-boot-starter-parent déclare un profile "native" qui configure l'exécution de la création d'une image native. Il suffit d'activer ce profile via l'option -P d'une ligne de commande.

mvn -Pnative spring-boot:build-image

image à ensuite lancer via docker run ...

Pour appronfondir le sujet :

<https://docs.spring.io/spring-boot/docs/current/reference/html/native-image.html>

1.1. Petit exemple (`native-image`)

Soit une toute petite application Spring Boot basée sur spring-Mvc avec une minuscule api REST (ex: `my-native-app` de <https://github.com/didier-mycontrib/spring6plus>).

NB: Début 2024, une première tentative avec une application springBoot complète (SpringWeb + Spring Data Jpa avec Hibernate) s'est soldée par un échec (problème temporaire d'incompatibilité entre Spring Native image et Hibernate pas encore résolu).

Vagrantfile

```
Vagrant.configure("2") do |config|
  config.vm.box = "bento/ubuntu-22.04"
  config.vm.network "private_network", ip: "192.168.33.10"
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "4096"
  end
  config.vm.provision "shell", path: "install-openjdk17or21-and-maven-on-ubuntu.sh"
  config.vm.provision :docker
  config.vm.provision :docker_compose
  config.vm.provision "shell", inline: <<-SHELL
    echo "verify docker and docker_compose ..."
    docker --version
    docker-compose --version
  SHELL
end
```

`install-openjdk17or21-and-maven-on-ubuntu.sh`

```
sudo apt-get update
sudo apt-get -y install openjdk-17-jdk
java -version
sudo apt-get -y install maven
mvn -version
```

```
cd /vagrant/git-repo
git clone https://github.com/didier-mycontrib/spring6plus
cd spring6plus/my-native-app
```

```
sh ./build_native_image.sh
#docker image rm my-native-app:0.0.1-SNAPSHOT
mvn -Pnative clean spring-boot:build-image
```

Lancement (via docker) de l'image native :

```
docker run --rm -d -p 8181:8181
  --name native-app-container my-native-app:0.0.1-SNAPSHOT
#docker stop native-app-container
```

Messages issus de la construction de l'image native :

```
GraalVM Native Image: Generating 'tp.mynativeapp.MyNativeAppApplication' (static executable)...
=====
```

```
[1/8] Initializing... (11.7s @ 0.14GB)
```

Java version: 17.0.10+13-LTS, vendor version: Liberica-NIK-23.0.3-1

Graal compiler: optimization level: 2, target machine: compatibility

C compiler: gcc (linux, x86_64, 11.4.0)

```
=====
```

```
[2/8] Performing analysis... [*****] (136.2s @ 1.30GB)
```

15,850 (90.51%) of 17,512 types reachable

26,423 (67.73%) of 39,011 fields reachable

77,502 (63.00%) of 123,019 methods reachable

5,006 types, 356 fields, and 5,777 methods registered for reflection

64 types, 70 fields, and 55 methods registered for JNI access

4 native libraries: dl, pthread, rt, z

```
[3/8] Building universe... (13.0s @ 1.59GB)
```

```
[4/8] Parsing methods... [***] (10.7s @ 1.79GB)
```

```
[5/8] Inlining methods... [***] (6.9s @ 1.79GB)
```

```
[6/8] Compiling methods... [*****] (101.9s @ 1.56GB)
```

```
[7/8] Layouting methods... [***] (7.7s @ 1.72GB)
```

```
[8/8] Creating image... [***] (11.0s @ 1.25GB)
```

38.39MB (50.83%) for code area: 50,627 compilation units

34.25MB (45.34%) for image heap: 379,149 objects and 255 resources

2.89MB (3.83%) for other data

75.54MB in total

```
=====
```

Successfully built image 'docker.io/library/my-native-app:0.0.1-SNAPSHOT'

BUILD SUCCESS **Total time: 06:40 min**

Consommations "mémoire" comparées*** En lancement via image docker ordinaire :**

	total	used	free
Mem:	4005976	399844	2335728 (après lancement)
Mem:	4005976	281696	2520400 (avant lancement)
<hr/>			
environ 120 Mo			

*** En lancement via image docker native :**

	total	used	free
Mem:	4005976	338120	2499476 (après lancement)
Mem:	4005976	299520	2610332 (avant lancement)
<hr/>			
seulement environ 40 Mo !!!			

XXIII - Annexe – énoncés des Tps / Spring

1. Tp sur bases de spring-framework

Récupérer une copie de https://github.com/didier-tp/spring_2025.git (via **git clone** ou via **code/download-zip** + extraction du zip dans **c:\tp** ou ailleurs)

1.1. (Tp facultatif) Très rapide aperçu sur ancienne config XML

L'ancienne configuration de Spring au format XML est aujourd'hui considérée comme obsolète. Elle est néanmoins toujours supportée par spring6 .
Ce Tp facultatif n'est à priori utile que si besoin de comprendre la structure d'une ancienne application Spring.

L'application exemple "*oldXmlSpringApp*" (au format "maven") du référentiel git https://github.com/didier-tp/spring_2025.git est un **exemple simple de configuration Spring XML** .

Il est possible de charger ce projet dans un IDE tel que eclipse ou intelliJ pour ensuite lancer l'application ou bien les tests unitaires .

La partie configuration XML se situe dans le sous répertoire **src/main/resources** .

1.2. Chargement du projet et analyses/vérifications

L'objectif de cette première série de Tps est d'appréhender les fonctionnalités essentielles de Spring via une approche très progressive en partant volontairement d'un début de projet uniquement basé sur **spring-framework** (sans spring-boot) .

Charger dans eclipse ou intelliJ le projet suivant (au format maven) :

tp/debutAppliSpringSansSpringBoot (*de https://github.com/didier-tp/spring_2025.git*)

NB: ce projet nécessite **java 17** comme version minimum du java .

Avec **intelliJ** , vérifier si besoin les réglages java/jdk via le menu "**file / project structure**"

- Analyser la structure de **pom.xml** (avec packaging="war")
- Repérer les principales dépendances (spring-context , ...)
- Lancer l'exécution de **tp.appliSpring.exemple.ExempleApp**
- Analyser le code initial de cet exemple simple
- Beaucoup de choses seront approfondies ultérieurement

1.3. Bases de l'injection de dépendance (avec @Autowired)

- dans package **tp.appliSpring.exemple** , Répéter la classe **Coordinateur** avec le début de code suivant (à compléter)

```

package tp.appliSpring.exemple;
//...
@Component
public class Coordinateur {
//...
private MonAfficheur monAfficheur=null; //référence vers afficheur à injecter
//...
private MonCalculateur monCalculateur=null;//référence vers calculateur à injecter
public void calculerEtAfficher() {
    double x=4;
    double res =monCalculateur.calculer(x); //x*x ou bien 2*x ou bien ...
    monAfficheur.afficher("res="+res);//>> res=16 en v1 ou bien ** res=16
}
}

```

- Au sein de la méthode main() de la classe ExempleApp , ajouter un bloc de code de ce genre :

```

Coordinateur coordinateurPrisEnChargeParSpring =
    contextSpring.getBean(Coordinateur.class);
coordinateurPrisEnChargeParSpring.calculerEtAfficher();

```

- Compléter le code de la classe Coordinateur (et ajuster si besoin d'autres classes) de manière à ce que cet exemple fonctionne bien.
- On pourra coder et tester successivement plein de variantes d'injection de dépendances :
 - via **@Autowired** (ou bien **@Resource** ou bien **@Inject**) sans ou avec affichage des éléments injectés au sein du constructeur par défaut de la classe Coordinateur et d'une méthode initialiser() préfixée par **@PostConstruct**
 - via des ajouts de **MonAfficheurV2** (avec préfixe "***" plutôt que ">>") et **MonCalculateurDouble** (2*x plutôt que x*x) de manière à engendrer une ambiguïté.
 - Via des ajouts de **@Qualifier** pour lever les ambiguïtés
 - Via une expérimentation de l'**injection par constructeur** (par exemple dans une classe "CoordinateurAvecInjectionParConstructeur")

1.4. Configurations via classes java (@Configuration, @Bean)

- Analyser la structure de la partie **tp.appliSpring.explicit.app.SpringAppWithExplicitConf** avec main()
 - .beans/ avec interfaces et classes sans aucune annotation spring
 - .conf.ExempleConfigExplicite à compléter en Tp

A faire en Tp:

phase 0: faire fonctionner l'exemple tel quel

phase 1: tenir compte des choix de prefixe et suffixe de exemples.properties via **@Value("\${...:...}")**

avec dans src/main/resources/exemples.properties

```
preferences.prefixe=>>>
preferences.suffixe=<<<
```

tester le comportement en modifiant les valeurs des prefixe et suffixe dans le .properties

phase 2: variante ...Basic en l'absence de profile "maj"

et variante ...Maj si présence du profile "maj"

tester le comportement en activant ou pas de profile "maj" en début de main()

NB: cette partie "*explicit*" est plus complexe que la partie "*exemple*" et n'a pas beaucoup d'intérêt tel quel .

Par contre, au sein d'un projet plus complexe, la configuration explicite basée sur @Bean peut s'avérer très utile pour paramétriser des composants "spring" basés sur des classes (récupérées via maven depuis une librairie externe) dont on n'a pas le droit de changer le code source .

1.5. Mise en place d'un aspect de type "log automatique"

Mettre en place un **aspect** (via Spring AOP , paramétré via annotations de AspectJ) qui affichera des lignes de logs pour chaque appel d'une méthode d'une classe du package

tp.appliSpring.exemple .

On pourra par exemple préciser le temps d'exécution et les noms des méthodes invoquées.

Code initial à compléter :

```
tp.appliSpring.annotation.LogExecutionTime
tp.appliSpring.aspect.MyPerfLogAspect
```

Phase 1 du Tp :

compléter (via ajout d'annotations et de code) les classes suivantes pour que l'aspect soit déclenché sur l'exécution des méthodes des classes du package tp.appliSpring.exemple lorsque le profile "perf" est activé :

```
tp.appliSpring.aspect.MyPerfLogAspect
tp.appliSpring.exemple.ExempleConfig
tp.appliSpring.exemple.ExempleApp.main()
```

Phase 2 du Tp :

Conditionner le comportement précédent au fait que l'annotation **@LogExecutionTime** doit être placée au dessus des méthodes sur lesquelles on souhaite obtenir un temps d'exécution .

2. Tp sur transaction spring et accès aux données (jdbc,jpa) sans spring-data

NB1 : cette série de Tps peut soit être effectuée pas à pas si l'on souhaite bien comprendre la structure des anciennes applications Spring (basées uniquement sur Spring-framework) . Si par contre, on préfère se contenter d'un très rapide aperçu pour la compréhension , on pourra directement analyser et faire fonctionner la solution (projet "appliSpringSansSpringBoot" sans "debut....") .

NB2 : le contenu initial du projet "*debutAppliSpringSansSpringBoot*" du référentiel maven https://github.com/didier-tp/spring_2025.git comporte une bonne partie des éléments des Tps ci-après → énoncés simplifiés (vérifier présence et contenu d'un fichier à améliorer plutôt que copier/coller) . En règle générale , le contenu initial des fichiers est incomplet (il faut ajouter des annotations et d'autres morceaux de code) .

2.1. Accès aux données (DataSource JDBC) , DAO

- Créer package *tp.appliSpring.core.entity*
- Créer classe *Compte.java*

```
package tp.appliSpring.core.entity;

public class Compte {

    private Long numero;
    private String label;
    private Double solde;

    //+get/set , constructeurs , toString()
}
```

- Créer package *tp.appliSpring.core.dao*
- Créer interface *DaoCompte.java*

```
package tp.appliSpring.core.dao;

import java.util.List;
import tp.appliSpring.core.entity.Compte;

public interface DaoCompte{
    Compte findById(Long numCpt);
    Compte save(Compte compte); //sauvegarde au sens saveOrUpdate
    List<Compte> findAll();
    void deleteById(Long numCpt);
    //...
}
```

- dans *src/main/resources* ajouter **application.properties** avec ce contenu :

```
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:~/mydbbank
spring.datasource.username=sa
spring.datasource.password=
```

- dans ***tp.appliSpring.core*** ajouter **MySpringApplication** avec ce contenu :

```
package tp.appliSpring.core;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

//version sans springBoot
@Configuration
@ComponentScan(basePackages = { "tp.appliSpring.core"})

//NB : Tous les sous packages de tp.appliSpring.core seront scrutés pour y découvrir
//@Component... et aussi pour y découvrir d'autres classes avec @Configuration
public class MySpringApplication {

    public static void main(String[] args) {
        //System.setProperty("spring.profiles.active", "p1");

        AnnotationConfigApplicationContext springContext = new
                AnnotationConfigApplicationContext(MySpringApplication.class);
        //...
        springContext.close();
    }
}
```

- Créer package ***tp.appliSpring.core.config***
- Créer la classe **CommonConfig.java**

```
package tp.appliSpring.core.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;

@Configuration
@PropertySource("classpath:/application.properties")
public class CommonConfig {

    @Bean
    public static PropertySourcesPlaceholderConfigurer
            propertySourcesPlaceholderConfigurer(){
        return new PropertySourcesPlaceholderConfigurer();
        //pour pouvoir interpréter ${} in @Value()
    }
}
```

- Créer la classe **DataSourceConfig.java**

```

package tp.appliSpring.core.config;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Configuration
public class DataSourceConfig {

    @Value("${spring.datasource.driverClassName}")
    private String jdbcDriver;

    @Value("${spring.datasource.url}")
    private String dbUrl;

    @Value("${spring.datasource.username}")
    private String dbUsername;

    @Value("${spring.datasource.password}")
    private String dbPassword;

    @Bean(name="dataSource")
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(jdbcDriver);
        dataSource.setUrl(dbUrl);
        dataSource.setUsername(dbUsername);
        dataSource.setPassword(dbPassword);
        return dataSource;
    }

    //seulement utile pour le dao en version Jdbc (avec NamedParameterJdbcTemplate):
    @Bean()
    public NamedParameterJdbcTemplate namedParameterJdbcTemplate( DataSource dataSource) {
        return new NamedParameterJdbcTemplate(dataSource);
    }
}

```

- dans *src/test/java* et dans un package **tp.appliSpring.dao** à créer, ajouter cette classe de test :

```

package tp.appliSpring.dao;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;

```

```

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import tp.appliSpring.core.MySpringApplication;
import tp.appliSpring.core.dao.DaoCompte;
import tp.appliSpring.core.entity.Compte;

@ExtendWith(SpringExtension.class) //si junit5/jupiter
@ContextConfiguration(classes= {MySpringApplication.class})
public class TestCompteDao {

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

    @Autowired
    @Qualifier("jdbc")
    //@Qualifier("jpa")
    private DaoCompte daoCompte; //à tester

@Test
public void testAjoutEtRelectureEtSuppression() {
    //hypothèse : base avec tables vides et existantes au lancement du test
    Compte compte = new Compte(null, "compteA", 100.0);
    Compte compteSauvegarde = this.daoCompte.save(compte); //INSERT INTO
    logger.debug("compteSauvegarde=" + compteSauvegarde);

    Compte compteRelu = this.daoCompte.findById(compteSauvegarde.getNumero());
    Assertions.assertEquals("compteA",compteRelu.getLabel());
    Assertions.assertEquals(100.0,compteRelu.getSolde());
    logger.debug("compteRelu apres insertion=" + compteRelu);

    compte.setSolde(150.0); compte.setLabel("compte_a");
    Compte compteMisAJour = this.daoCompte.save(compte); //UPDATE
    logger.debug("compteMisAJour=" + compteMisAJour);

    compteRelu = this.daoCompte.findById(compteSauvegarde.getNumero()); //SELECT
    Assertions.assertEquals("compte_a",compteRelu.getLabel());
    Assertions.assertEquals(150.0,compteRelu.getSolde());
    logger.debug("compteRelu apres miseAJour=" + compteRelu);

    //+supprimer :
    this.daoCompte.deleteById(compteSauvegarde.getNumero());

    //vérifier bien supprimé (en tentant une relecture qui renvoi null)
    Compte compteReluApresSuppression =
        this.daoCompte.findById(compteSauvegarde.getNumero());
    Assertions.assertTrue(compteReluApresSuppression == null);
}
}

```

Script de préparation de la base de données (ici en version H2) :

init_db.sql

```
DROP TABLE IF EXISTS Compte;

CREATE TABLE Compte(
    numero integer auto_increment NOT NULL,
    label VARCHAR(64),
    solde double,
    PRIMARY KEY(numero));

INSERT INTO Compte (label,solde) VALUES ('compte courant',100);
INSERT INTO Compte (label,solde) VALUES ('compte codevi',50);
INSERT INTO Compte (label,solde) VALUES ('compte 3',150);

SELECT * FROM Compte;
```

set_env.bat

```
set MVN_REPOSITORY=C:\Users\administrateur\.m2\repository
set MY_H2_DB_URL=jdbc:h2:~/mydbbank
set H2_VERSION=2.2.224
set H2_CLASSPATH=%MVN_REPOSITORY%\com\h2database\h2\%H2_VERSION%\h2-%H2_VERSION%.jar
```

create_h2_database.bat

```
cd /d %~dp0
call set_env.bat
java -classpath %H2_CLASSPATH% org.h2.tools.RunScript -url %MY_H2_DB_URL% -user sa -script init_db.sql -showResults
pause
```

lancer_console_h2.bat

```
cd /d %~dp0
call set_env.bat
java -jar %H2_CLASSPATH% -user "sa" -url %MY_H2_DB_URL%
REM NB: penser à se déconnecter pour éviter des futurs verrous/blocages
pause
```

NB : Toute cette structure de code et configuration sera utilisée dès le(s) TP(s) suivant(s)

2.2. Petit exemple de DAO via JDBCTemplate

NB: Ce TP pas fondamental est facultatif : à faire ou pas selon le temps disponible

coder le début de ***DaoCompteJdbc.java*** avec le code suivant (*à compléter*)

```
package tp.appliSpring.core.dao;

import java.sql.ResultSet; import java.sql.SQLException;
import java.util.HashMap; import java.util.List; import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
```

```

import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;
import tp.appliSpring.core.entity.Compte;

@Repository // @Component de type DAO/Repository
@Qualifier("jdbc")
public class DaoCompteJdbc /*extends JdbcDaoSupport*/ implements DaoCompte {

private final String INSERT_SQL = "INSERT INTO compte(label, solde) values(:label,:solde)";
private final String UPDATE_SQL = "UPDATE compte set label=:label , solde=:solde where numero=:numero";
private final String FETCH_ALL_SQL = "select * from compte";
private final String FETCH_BY_NUM_SQL = "select * from compte where numero=:numero";
private final String DELETE_BY_NUM_SQL = "delete from compte where numero=:numero";

@Autowired
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

@Override
public Compte findById(Long numCpt) {
    Compte compte = null;
    Map<String, Long> parameters = new HashMap<String, Long>();
    parameters.put("numero", numCpt);
    List<Compte> comptes = namedParameterJdbcTemplate.query(FETCH_BY_NUM_SQL,
                                                                parameters, new CompteMapper());
    compte = comptes.isEmpty()?null:comptes.get(0);
    return compte;
}

@Override
public Compte save(Compte compte) {
    if(compte==null)
        throw new IllegalArgumentException("compte must be not null");
    return (compte.getNumero()==null)?insert(compte):update(compte);
}

public Compte insert(Compte compte) {
    KeyHolder holder = new GeneratedKeyHolder(); //to retreive auto_increment value of pk
    SqlParameterSource parameters = new MapSqlParameterSource()
        .addValue("label", compte.getLabel())
        .addValue("solde", compte.getSolde());
    namedParameterJdbcTemplate.update(INSERT_SQL, parameters, holder);
    compte.setNumero(holder.getKey().longValue()); //store auto_increment pk in instance to return
    return compte;
}

public Compte update(Compte compte) {
    //A CODER/COMPLETER EN TP
}

@Override
public List<Compte> findAll() {
    //A CODER/COMPLETER EN TP
}

```

```

@Override
public void deleteById(Long numCpt) {
    //A CODER/COMPLÉTER EN TP
}

}

//classe auxiliaire "CompteMapper" pour convertir Resultset jdbc en instance de la classe Compte :
class CompteMapper implements RowMapper<Compte> {
    @Override
    public Compte mapRow(ResultSet rs, int rowNum) throws SQLException {
        Compte compte = new Compte();
        compte.setNumero(rs.getLong("numero"));
        compte.setLabel(rs.getString("label"));
        compte.setSolde(rs.getDouble("solde"));
        return compte;
    }
}

```

- Compléter le code manquant de cette classe
- tester via le lancement de **TestCompteDao** (dans src/test/java)

2.3. Accès aux données via JPA/Hibernate

Ajouter dans le package **tp.appliSpring.core.config** la classe de configuration **DomainAndPersistenceConfig.java** suivante :

```

package tp.appliSpring.core.config;

import java.util.Properties; import javax.sql.DataSource;
import jakarta.persistence.EntityManagerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.Database;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@EnableTransactionManagement() // "transactionManager" (not "txManager") is expected !!!
@ComponentScan(basePackages = { "tp.appliSpring.core.dao" ,
                                "tp.appliSpring.core.service" , "tp.appliSpring.core.init"})
```

public class **DomainAndPersistenceConfig** {

```

// JpaVendorAdapter (Hibernate ou OpenJPA ou ...)
@Bean
public JpaVendorAdapter jpaVendorAdapter() {
    HibernateJpaVendorAdapter hibernateJpaVendorAdapter =
        new HibernateJpaVendorAdapter();
    hibernateJpaVendorAdapter.setShowSql(false);
    hibernateJpaVendorAdapter.setGenerateDdl(false);
}
```

```

//hibernateJpaVendorAdapter.setDatabase(Database.MYSQL);
hibernateJpaVendorAdapter.setDatabase(Database.H2);
return hibernateJpaVendorAdapter;
}

// EntityManagerFactory
@Bean(name = { "entityManagerFactory" })
public EntityManagerFactory entityManagerFactory(JpaVendorAdapter jpaVendorAdapter,
                                                DataSource dataSource) {
    LocalContainerEntityManagerFactoryBean factory =
        new LocalContainerEntityManagerFactoryBean();
    factory.setJpaVendorAdapter(jpaVendorAdapter);
    factory.setPackagesToScan("tp.appliSpring.core.entity");
    factory.setDataSource(dataSource);

    Properties jpaProperties = new Properties(); //java.util
    jpaProperties.setProperty("javax.persistence.schema-generation.database.action",
                             "drop-and-create"); //JPA>=2.1
    factory.setJpaProperties(jpaProperties);
    factory.afterPropertiesSet();
    return factory.getObject();
}

// Transaction Manager for JPA or ...
@Bean(name = "transactionManager")
public PlatformTransactionManager transactionManager(
    EntityManagerFactory entityManagerFactory) {
    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory);
    return txManager;
}
}

```

Coder au sein du package ***tp.appliSpring.core.dao*** la classe ***DaoCompteJpa*** en partant du code suivant (à compléter) :

```

package tp.appliSpring.core.dao;

import java.util.List;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import tp.appliSpring.core.entity.Compte;

@Repository // @Component de type DAO/Repository
@Qualifier("jpa")
public class DaoCompteJpa implements DaoCompte {

    @PersistenceContext
    private EntityManager entityManager;

    @Override

```

```

public Compte findById(Long numCpt) {
    // A CODER/COMPLÉTER EN TP
}

/*
public Compte save(Compte compte) {
    try {
        entityManager.beginTransaction().begin();
        if(compte.getNumero()==null)
            entityManager.persist(compte);//INSERT INTO
        else
            entityManager.merge(compte); //UPDATE
        entityManager.getTransaction().commit();
    } catch (Exception e) {
        entityManager.getTransaction().rollback();
        e.printStackTrace();
    }
    return compte; //avec numero plus null (auto_incrémenté)
}
*/

```

@Override
@Transactional

```

public Compte save(Compte compte) {
    if(compte.getNumero()==null)
        entityManager.persist(compte); //INSERT INTO
    else
        entityManager.merge(compte); //UPDATE
    return compte; //avec numero plus null (auto_incrémenté)
}
```

@Override

```

public List<Compte> findAll() {
    return entityManager.createQuery("SELECT c FROM Compte c",
                                    Compte.class)
        .getResultSet();
}
```

@Override
@Transactional

```

public void deleteById(Long numCpt) {
    // A CODER/COMPLÉTER EN TP
    Compte compte = .....
    entityManager.....(compte);
}
```

- Compléter le code de la classe ci-dessus
- Ajouter toutes les **annotations** manquantes et nécessaires dans la classe
`tp.appliSpring.core.entity.Compte (@Entity, @Id ,, @GeneratedValue(strategy = GenerationType.IDENTITY))`
- switcher de qualificatif
`@Qualifier("jdbce") Qualifier("jpa")` au sein de la classe **TestCompteDao**
- Lancer le test et corriger les éventuels problèmes/erreurs .

2.4. Service Spring et gestion des transactions

Créer le nouveau package `tp.appliSpring.core.service`

Ajouter y l'interface `ServiceCompte` suivante :

```
package tp.appliSpring.core.service;

import java.util.List;
import tp.appliSpring.core.entity.Compte;

public interface ServiceCompte {
    Compte rechercherCompteParNumero(long numero);
    List<Compte> rechercherTousComptes();
    List<Compte> rechercherComptesDuClient(long numClient);
    Compte sauvegarderCompte(Compte compte);
    void supprimerCompte(long numCpt);
    void transferer(double montant, long numCptDeb, long numCptCred);
}
```

Ajouter la classe d'implémentation `ServiceCompteImpl` suivante (*à compléter*) :

```
package tp.appliSpring.core.service;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import tp.appliSpring.core.dao.DaoCompte;
import tp.appliSpring.core.entity.Compte;

@Service //classe de Service prise en charge par spring
public class ServiceCompteImpl implements ServiceCompte {

    @Qualifier("jpa")
    @Autowired
    private DaoCompte daoCompte=null;

    public Compte rechercherCompteParNumero(long numero) {
        return daoCompte.findById(numero);
    }

    public Compte sauvegarderCompte(Compte compte) {
        return daoCompte.save(compte);
    }

    public List<Compte> rechercherTousComptes() {
        // A CODER/COMPLÉTER EN TP
    }

    public List<Compte> rechercherComptesDuClient(long numClient) {
        //return null; //version zero
    }
}
```

```

        return this.rechercherTousComptes(); //V1 (provisoire)
        //future version V2 (via un nouvel appel sur DAO exploitant @ManyToOne ou bien ...)
    }

    public void supprimerCompte(long numCpt) {
        // A CODER/COMPLETER EN TP
    }

    @Transactional(*propagation = Propagation.REQUIRED*) //REQUIRED par defaut
    public void transferer(double montant, long numCptDeb, long numCptCred) {
        try {
            // transaction globale initialisée dès le début de l'exécution de transferer
            Compte cptDeb = this.daoCompte.findById(numCptDeb);
            //le dao exécute son code dans la grande transaction
            //commencée par le service sans la fermer et l'objet cptDeb remonte à l'état persistant
            cptDeb.setSolde(cptDeb.getSolde() - montant);
            //this.daoCompte.save(cptDeb); //facultatif si @Transactional

            //idem pour compte à créditer
            Compte cptCred= this.daoCompte.findById(numCptCred);
            cptCred.setSolde(cptCred.getSolde() + montant);
            //this.daoCompte.save(cptCred) //facultatif si @Transactional

            //en fin de transaction réussie (sans exception) , toutes les modification effectuées
            //sur les objets à l'état persistant seront répercutées en base (.save() automatiques)
        } catch (Exception e) {
            throw new RuntimeException("echec virement " + e.getMessage(), e);
            //rollback se fait de façon fiable
            //ou bien throw new
            //ClasseExceptionPersonnaliseeHeritantDeRuntimeException("echec virement", e);
        }
    }
}

```

Au sein de `src/test/java` et du package `tp.appliSpring.core.service` (à créer), ajouter la classe de test `TestServiceCompte` suivante :

```
package tp.appliSpring.service;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import tp.appliSpring.core.MySpringApplication;
import tp.appliSpring.core.entity.Compte;
import tp.appliSpring.core.service.ServiceCompte;

@ExtendWith(SpringExtension.class)
```

```

@ContextConfiguration(classes= {MySpringApplication.class})
//{@ActiveProfiles({ "embeddedDB", "dev", "perf" })
public class TestServiceCompte {

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

@Autowired
private ServiceCompte serviceCompte; //à tester

@Test
public void testVirement() {
    Compte compteASauvegarde = this.serviceCompte.sauvegarderCompte(
        new Compte(null,"compteA",300.0));
    Compte compteBSauvegarde = this.serviceCompte.sauvegarderCompte(
        new Compte(null,"compteB",100.0));
    long numCptA = compteASauvegarde.getNumero();
    long numCptB = compteBSauvegarde.getNumero();
    //remonter en memoire les anciens soldes des compte A et B avant virement
    //(+affichage console ou logger) :
    double soldeA_avant= compteASauvegarde.getSolde();
    double soldeB_avant = compteBSauvegarde.getSolde();
    logger.debug("avant bon virement, soldeA_avant="+soldeA_avant +
        " et soldeB_avant=" + soldeB_avant);

    //effectuer un virement de 50 euros d'un compte A vers vers compte B
    this.serviceCompte.transferer(50.0, numCptA, numCptB);

    //remonter en memoire les nouveaux soldes des compte A et B apres virement
    // (+affichage console ou logger)
    Compte compteAReluApresVirement =
        this.serviceCompte.rechercherCompteParNumero(numCptA);
    Compte compteBReluApresVirement =
        this.serviceCompte.rechercherCompteParNumero(numCptB);
    double soldeA_apres = compteAReluApresVirement.getSolde();
    double soldeB_apres = compteBReluApresVirement.getSolde();
    logger.debug("apres bon virement, soldeA_apres="+soldeA_apres +
        " et soldeB_apres=" + soldeB_apres);

    //verifier -50 et +50 sur les différences de soldes sur A et B :
    Assertions.assertEquals(soldeA_avant - 50, soldeA_apres,0.000001);
    Assertions.assertEquals(soldeB_avant + 50, soldeB_apres,0.000001);
}

//@Test
public void testMauvaisVirement() {
    /* VARIANTE A CODER/COLPLETER EN TP
    COPIER/COLLER à ADAPTER de testVirement()
    AVEC
    try {
        this.serviceCompte.transferer(50.0, numCptA, -numCptB); //erreur volontaire
    } catch (Exception e) {
        logger.error("echec normal du virement " + e.getMessage());
    }
}

```

```
    }
    et
    //vérifier -0 et +0 sur les différences de soldes sur A et B
    Assertions.assertEquals(soldeA_avant, soldeA_apres, 0.000001);
    Assertions.assertEquals(soldeB_avant, soldeB_apres, 0.000001);
    */
}
}
```

Série de tests à effectuer :

1. enlever `@Transactional` au dessus de la méthode **transferer** et
enlever les commentaires sur les lignes `this.daoCompte.save(cptDeb);`
`et this.daoCompte.save(cptCred);`
2. lancer le test **testVirement()** et corriger les bugs si nécessaire
3. coder et lancer **testMauvaisVirement()** . c'est normal si ça ne fonctionne pas bien sans
l'ajout de `@Transactional`
4. replacer `@Transactional` au dessus de la méthode **transferer** et relancer le test
testMauvaisVirement() qui devrait normalement fonctionner .
5. replacer des commentaires sur les lignes `this.daoCompte.save(cptDeb);`
`et this.daoCompte.save(cptCred);`
Tous les tests devraient encore bien fonctionner .

3. Tp sur transactions , spring-data (avec spring-boot)

Récupérer une copie de https://github.com/didier-tp/spring_2025.git (via **git clone** ou via **code/download-zip** + extraction du zip dans **c:\tp** ou ailleurs)

3.1. (Tp facultatif) , nouveau projet via spring initializr

- Aller sur le site "spring initializr"
- Créer un nouveau projet ("demo" , "java/maven/java_17" , "jar") avec les dépendances fondamentales ("jpa" , "web" , ...)
- extraire le contenu de demo.zip dans **d:\tp** ou ailleurs
- charger ce projet dans eclipse ou intelliJ
- ajouter un numéro de port dans application.properties
- ajouter une page index.html au sein de src/main/resources/static
- démarrer l'application (via le main()) et vérifier l'accès à la page d'accueil

NB: avec beaucoup de temps, on pourrait effectuer tous les Tps ultérieurs à partir de ce nouveau projet .

Pour gagner un peu de temps, les Tps ultérieurs seront effectués à partir d'un point de départ plus riche et bien structuré tp/debutAppliSpringWeb .

Néanmoins, pour les "débutants Spring" , enrichir petit à petit le point de départ "demo" peut être un exercice complémentaire intéressant si l'on souhaite bien comprendre toutes les configurations nécessaires au bon fonctionnement d'une application springBoot .

3.2. Chargement du projet springBoot et analyses/vérifications

L'objectif de cette seconde série de Tps est d'appréhender les fonctionnalités essentielles de Spring en mode "configuration moderne" (avec spring-boot et spring-data) .

Charger dans eclipse ou intelliJ le projet suivant (au format maven) :

tp/debutAppliSpringWeb (de https://github.com/didier-tp/spring_2025.git)

NB: ce projet nécessite **java 17** comme version minimum du java .

Avec **intelliJ** , vérifier si besoin les réglages java/jdk via le menu "**file / project structure**"

- Analyser la structure de **pom.xml** (avec packaging="jar")
- Repérer les principales dépendances (spring-boot-starter-...)
- Lancer l'exécution de **tp.appliSpring.AppliSpringApplication.main()**
- Visualiser la page d'accueil au sein d'un navigateur
(<http://localhost:8181/appliSpring>)
partie fonctionnant dès le début : compte1 au format json
- Beaucoup de choses seront approfondies ultérieurement

3.3. Analyse de application.properties et variantes (profiles)

NB :

- Le fichier **application.properties** comporte les configurations essentielles (n° port, ...)
- Le fichier **application-dev.properties** (associé au profil "dev") comporte une configuration vers une base de données "h2" (embedded database sans serveur) dont l'url est **jdbc:h2:~/mydbbank**
- **Les tables de cette base de données seront re-crées (à vide) à chaque redémarrage** (du main() ou des test avec le profil "dev") du fait de la propriété **spring.jpa.hibernate.ddl-auto=create**
- D'autres profils (ex : "prod") comporte **spring.jpa.hibernate.ddl-auto=none**

3.4. Dao en mode JpaRepository

Analyser les portions de code suivantes :

tp.appliSpring.bank.persistence.entity.CompteEntity
 tp.appliSpring.bank.persistence.repository.CompteRepository

Faire fonctionner le test suivant :

tp.appliSpring.core.dao.TestCompteDao

Phase1 du Tp :

Ajouter dans CompteRepository une méthode permettant de rechercher les comptes dont le solde est entre une valeur mini et une valeur maxi.

On proposera deux versions de cette méthode :

- une première respectant les convention de nommage des méthodes pour une requête automatique
- une seconde codée avec @Query() et JpaQL

On testera ceci en ajoutant une méthode de test au sein de TestCompteDao

Phase2 (facultative) du Tp :

- Ajouter dans OperationRepository une méthode permettant de rechercher les opérations rattachées à un numéro de compte précis et dont la date est entre une valeur mini et une valeur maxi (les paramètres dateMini , dateMaxi seront de type "Date")
- On testera ceci en ajoutant une méthode de test au sein de TestOperationDao (à coder à coté de TestCompteDao) avec par exemple SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd"); et **dateFormat.parse("2025-02-12")**

Phase3 (facultative) du Tp :

- coder une variante (ex : TestOperationDaoSql) de la classe TestOperationDao avec l'utilisation de @Sql pour injecter des données en base en début de méthode de test

Phase4 (facultative) du Tp :

- expérimentations libres (sans casser ce qui fonctionne dès le départ)

3.5. Transactions sur virement bancaire

Analyser globalement la structure de code suivante :

- tp.appliSpring.bank.core.model.Compte
- tp.appliSpring.bank.core.service.ServiceCompte
- tp.appliSpring.bank.core.service.direct.ServiceCompteDirectImpl

Code important de la classe tp.appliSpring.bank.core.service.direct.ServiceCompteDirectImpl :

```
// ...
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
// ...

@Service //classe de Service prise en charge par spring
@Transactional
@Qualifier("direct")
@Primary
public class ServiceCompteDirectImpl .... implements ServiceCompte{

// ...

@Transactional/*propagation = Propagation.REQUIRED*/ //REQUIRED par defaut
public void transfer(double montant, long numCptDeb, long numCptCred)
    throws BankException{
try {
    // transaction globale initialisée dès le début de l'exécution de transferer
    CompteEntity cptDeb = this.daoCompte.findById(numCptDeb).get();
        //le dao exécute son code dans la grande transaction
        //commencée par le service sans la fermer et l'objet cptDeb remonte à l'état persistant
    cptDeb.setSolde(cptDeb.getSolde() - montant);
    this.daoCompte.save(cptDeb); //facultatif si @Transactional

    //idem pour compte à créditer
    CompteEntity cptCred= this.daoCompte.findById(numCptCred).get();
    cptCred.setSolde(cptCred.getSolde() + montant);
    this.daoCompte.save(cptCred) ; //facultatif si @Transactional

    //en fin de transaction réussie (sans exception) , toutes les modification effectuées
    //sur les objets à l'état persistant seront répercutées en base (.save() automatiques)
} catch (Exception e) {
    throw new BankException("echec virement",e);
    //le rollback se fait de façon fiable car BankException hérite de RuntimeException
}
}
```

Au sein de *src/test/java* et du package *tp.appliSpring.core.service* , visualiser la classe de test *TestServiceCompte* avec la partie de code importante suivante :

```

// ...
@SpringBootTest(classes= {AppliSpringApplication.class})
@ActiveProfiles({ "dev" })
public class TestServiceCompte {

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

    @Autowired
    private ServiceCompte serviceCompte; //à tester

    @Test
    public void testVirement() {
        Compte compteASauvegarde = this.serviceCompte.create(
            new Compte(null,"compteA",300.0));
        Compte compteBSauvegarde = this.serviceCompte.create(
            new Compte(null,"compteB",100.0));
        long numCptA = compteASauvegarde.getNumero();
        long numCptB = compteBSauvegarde.getNumero();
        //remonter en memoire les anciens soldes des compte A et B avant virement
        //(+affichage console ou logger) :
        double soldeA_avant= compteASauvegarde.getSolde();
        double soldeB_avant = compteBSauvegarde.getSolde();
        logger.debug("avant bon virement, soldeA_avant="+soldeA_avant +
                    " et soldeB_avant=" + soldeB_avant);

        //effectuer un virement de 50 euros d'un compte A vers vers compte B
        this.serviceCompte.transfer(50.0, numCptA, numCptB);

        //remonter en memoire les nouveaux soldes des compte A et B apres virement
        // (+affichage console ou logger)
        Compte compteAReluApresVirement =
            this.serviceCompte.searchById(numCptA);
        Compte compteBReluApresVirement =
            this.serviceCompte.searchById(numCptB);
        double soldeA_apres = compteAReluApresVirement.getSolde();
        double soldeB_apres = compteBReluApresVirement.getSolde();
        logger.debug("apres bon virement, soldeA_apres="+soldeA_apres +
                    " et soldeB_apres=" + soldeB_apres);

        //verifier -50 et +50 sur les differences de soldes sur A et B :
        Assertions.assertEquals(soldeA_avant - 50, soldeA_apres,0.000001);
        Assertions.assertEquals(soldeB_avant + 50, soldeB_apres,0.000001);
    }

    //@Test
    public void testMauvaisVirement() {
        /* VARIANTE A CODER/COMPLETER EN TP
        COPIER/COLLER à ADAPTER de testVirement()
        AVEC
        try {
            this.serviceCompte.transfer(50.0, numCptA, -numCptB); //erreur volontaire
        } catch (Exception e) {
        */
    }
}

```

```
        logger.error("echec normal du virement " + e.getMessage());
    }
    et
    //vérifier -0 et +0 sur les différences de soldes sur A et B
    Assertions.assertEquals(soldeA_avant, soldeA_apres,0.000001);
    Assertions.assertEquals(soldeB_avant, soldeB_apres,0.000001);
    */
}
}
```

Série de tests à effectuer :

1. enlever `@Transactional` au dessus de la méthode **transfer**
2. lancer le test **testVirement()** et corriger les bugs si nécessaire
3. coder et lancer **testMauvaisVirement()** . c'est normal si ça ne fonctionne pas bien sans l'ajout de `@Transactional`
4. replacer `@Transactional` au dessus de la méthode **transfer** et relancer le test **testMauvaisVirement()** qui devrait normalement fonctionner .
5. Autres expérimentations libres (sans casser le code de départ)

4. Tp sur Api-REST avec Spring-Mvc

Série de Tps basée sur le point de départ **tp/debutAppliSpringWeb**

4.1. Familiarisation avec la structure du projet

Analysier globalement la structure suivante :

- tp.appliSpring.bank.core.model.Compte
- tp.appliSpring.bank.core.service.ServiceCompte
- tp.appliSpring.bank.core.init.ReInitDefaultDataSet avec profil "reInit"
- tp.appliSpring.generic.converter.GenericMapper
- tp.appliSpring.generic.dto.ApiError
- tp.appliSpring.generic.dto.MessageDto
- tp.appliSpring.generic.exception.EntityNotFoundException
- tp.appliSpring.bank.web.api.rest.CompteRestCtrl

4.2. Partie "get" de l'api REST

- Analyser le code initial de
tp.appliSpring.bank.web.api.rest.CompteRestCtrl
- Lancer l'application AppliSpringApplication.main()
avec les profils "dev" et "reInit"
- tester via la partie "compte1 au format JSON" de index.html
(sur http <http://localhost:8181/appliSpring>)
- coder la partie //GET Multiple de CompteRestCtrl
- tester via les liens hypertextes présents du haut de index.html

4.3. Gestion des statuts Http et des exceptions

- Déclencher l'url <http://localhost:8181/appliSpring/rest/api-bank/v1/comptes/18888> sachant que le compte 18888 n'existe pas et visualiser un message d'erreur pas très précis (500)
- Coder et tester la version V2 de CompteRestCtrl.getCompteById en s'appuyant sur *ResponseEntity<?>* de manière à renvoyer proprement le code 404/NOT_FOUND.
- Revenir sur la V1 de CompteRestCtrl.getCompteById et tester le comportement en décommentant @ControllerAdvice au sein de
tp.appliSpring.generic.rest.RestResponseEntityExceptionHandler
- Redémarrer et tester le nouveau comportement et le message d'erreur approprié
NB : utiliser la console du navigateur pour visualiser le code de retour HTTP 404.

4.4. Partie "post,put,delete" de l'api REST

Coder et tester progressivement les parties "POST, DELETE , PUT" de CompteRestController et tester cela via "[Try it out](#)" de la partie "[documentation swagger3/openapi](#)" .

On pourra éventuellement effectuer les tests avec un outils de type "postman" ou autre .

Bien tester les cas d'erreur .

Lorsque le code java de la classe CompteRestController aura été bien complété et bien testé , on pourra utiliser le mini front-end "**compteAjax**" ([html](#) + [js](#)) incorporé dans la partie **static** du projet de manière à invoquer l'api REST sur les comptes en mode CRUD .

4.5. Validation des entrées avec @Valid

On pourra retoucher/améliorer la méthode en mode "POST" de la classe CompteRestController en utilisant le type `tp.appliSpring.bank.web.api.dto.CompteToCreate` plutôt que `Compte`.

- La classe `CompteToCreate` comporte déjà des annotations de type `@Min`, `@Length`
- La classe `tp.appliSpring.generic.rest.RestResponseEntityExceptionHandler` comporte déjà la méthode `handleMethodArgumentNotValid`
- plus qu'à utiliser `@Valid` à coté de `@RequestBody` `CompteToCreate`
- bien tester le comportement avec des données valides ou pas

4.6. Test unitaire pour Api REST (tp facultatif)

Compléter la classe de test `TestCompteRestControllerWithServiceMockWithoutSecurity`

permettant de tester une partie des méthodes de `CompteRestController` .

Suggestions :

- `@WebMvcTest(controllers = { CompteRestController.class }, excludeAutoConfiguration = { SecurityAutoConfiguration.class })` ou autre
- `@Autowired private MockMvc mvc;`
- `mvc.perform(get("/rest/api-bank/v1/comptes?numClient=1"))` ou autre

4.7. Utilisation de mapStruct (tp facultatif)

- Décommenter tous les blocs liés à `mapStruct` dans `pom.xml`
- Réactiver l'interface `tp.appliSpring.bank.converter.MyBankMapper` en supprimant l'extention `.txt` sur le fichier)
- Relancer un build maven (ex : `mvn package`) et visualiser le code généré dans la partie

"target/generated-sources/annotations".

- Utiliser le convertisseur **MyBankMapper** plus performant que *GenericMapper* au sein des méthodes *searchWithMinimumBalance* et *searchCustomerAccounts* de la classe *tp.appliSpring.bank.core.service.direct.ServiceCompteDirectImpl*.
- Tester le bon fonctionnement après un redémarrage .

4.8. Exemple d'appel REST externe (tp facultatif)

Dans une nouvelle classe de type *util.client.MyWsCall* , utiliser l'ancienne api *RestTemplate* ou bien la nouvelle api *RestClient* de manière à invoquer un WS REST externe. On testera cela via une petite classe de test .

Exemple de WS-REST facile à appeler :

<http://api.zippopotam.us> (/fr/75001)

à appeler en mode GET

<https://www.d-defrance.fr/tp/devise-api/v1/public> /devises/EUR

à appeler en mode GET ou DELETE

<https://www.d-defrance.fr/tp/devise-api/v1/public> /devises

à appeler en mode GET ou POST

5. Tp sur DHTML via SpringMvc et Thymeleaf (ou JSP)

5.1. Analyse des exemples basiques et des templates thymeleaf

Exemples élémentaires :

- tp.appliSpring.bank.site.controller.BasicController (helloworld et calculTva)
- src/main/resources/templates (displayBasicMessage.html , calculTva.html)

Point d'entrée pour tester l'exécution de ces exemples :

index.html → index-site.html → ...

Avec "layout" de Thymeleaf :

- src/main/resources/templates/_footer.html , _header.html , _layout.html
- src/main/resources/templates/welcome.html , calcul_racine.html

5.2. Tp @SessionAttribute sur calcul de racine carrée

Par défaut , chaque calcul de racine carrée est indépendant.

welcome → calculRacineCarree → welcome → calculRacineCarree (avec 0.0 par défaut).

Si l'on souhaite expérimenter une mémorisation en session du dernier calcul de racine carrée effectué on pourra ajouter **@SessionAttributes(...)** et ce qui va avec au sein de tp.appliSpring.bank.site.controller.BasicController .

Vérifier le nouveau comportement via une navigation de ce type :

welcome → calculRacineCarree → welcome → calculRacineCarree (autre que 0.0).

5.3. Analyse d'autres exemples et expérimentations

- Partie "inscription" de tp.appliSpring.bank.site.controller.BasicController et tp.appliSpring.bank.site.form.InscriptionForm et src/main/resources/templates/inscription.html
pour exemple avec plein de syntaxes thymeleaf .
-

5.4. TP Virement bancaire via SpringMVC et Thymeleaf

- Navigation préalable à expérimenter :
index → index-site → welcome → espace_client (numClient=1) → comptesDuClient
puis comptesDuClient → virement
- Visualiser le code de tp.appliSpring.bank.site.form.**VirementForm**
- compléter src/main/resources/templates/**virement.html**
et tp.appliSpring.bank.site.controller.**BankController**
de manière à déclencher un virement bancaire entre des comptes du client connecté
(avec numClient et client en session)

Suggestions :

- Au sein de virement.html on pourra saisir ou choisir les parties montant , numCptDeb et numCptCred en tant que sous parties de th:object="\${virement}" via des syntaxes de type th:field="*{montant}" et envoyer le tout vers th:action="@{/site/bank/doVirement}" au sein d'un formulaire à déclencher en mode POST
- Au sein d'un nouveau point d'entrée de BankController (ex : "doVirement" on pourra récupérer les valeurs saisies via un paramètre d'entrée de type @ModelAttribute("virement") VirementForm virement puis déclencher un appel à serviceCompte.transfer(...) En cas de succès on pourra retourner comptesDuClient(model); //pour réactualiser et afficher nouvelle liste des comptes du client

6. Tp sur Spring-security

6.1. A savoir avec spring-boot-starter-security

En Ajoutant seulement ceci dans **pom.xml**

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

et redémarrant une application vierge sans aucune configuration.

La console affiche alors un message de ce genre :

Using generated security password: 656a4d96-2f13-46fd-b4b9-2e2c90b3fbb6

This generated password is for development use only. Your security configuration must be updated before running your application in production.

Ce mot de passe (régénéré à chaque démarrage) est pour le username "**user**"

Après avoir saisi "user" et le mot de passe attendu par copier/coller dans la boîte de dialogue , on peut accéder à la page d'accueil principale (index.html) et naviguer vers d'autres pages .

Autrement dit , la sécurité "spring" n'est réellement exploitable qu'avec un minimum de configuration et sans aucune configuration il vaut mieux de pas ajouter `spring-boot-starter-security` dans pom.xml .

6.2. withoutSecurity

Si en début de développement on souhaite (malgré la dépendance directe ou indirecte `spring-boot-starter-security`) utiliser la partie web de l'application en mode "sécurité désactivée" on peut utiliser une classe de configuration ressemblant à `tp.appliSpring.WithoutSecurityConfig` avec entre autres `auth.requestMatchers("/**").permitAll()` .

C'est dans ce mode de fonctionnement que les Tps des chapitres précédents ont été effectués.
Avec profils "dev,reInit" sélectionné au démarrage de l'application.

6.3. Sécurité sur Api REST en mode OAuth2/OIDC

- Repérer la dépendance fondamentale `spring-boot-starter-oauth2-resource-server` déjà placée dans pom.xml
- Repérer le paramétrage suivant au sein du fichier **application-withSecurity.properties**
`spring.security.oauth2.resourceserver.jwt.issuer-uri=`
`https://www.d-defrance.fr/keycloak/realmns/sandboxrealm`
- Compléter la partie manquante du paramétrage de la sécurité au sein du fichier `tp.appliSpring.security.SecurityConfigForRest`
A ce stade du Tp , la partie à compléter se situe au niveau de la sous fonction

```
private HttpSecurity restFilterChainBuilder(HttpSecurity http) appelée par le point d'entrée  

@Bean @Order(1) @Profile("!withoutOAuth2")  

    public SecurityFilterChain restFilterChainWithOAuth2(HttpSecurity http) .
```

- Démarrer l'application SpringBoot avec les profils "dev,reInit,withSecurity"
- Tester le comportement sécurisé de la partie rest via :
 - une première tentative de **compteAjax** avec un ajout de compte (label="ccc" , solde=0) menant normalement au message d'erreur 401/Unauthorized
 - une seconde tentative après un login oAuth2/oidc via le compte "admin1/pwd1"

6.4. Sécurité sur Api REST sans OAuth2 avec gestion directe JWT

- Repérer la partie **passwordEncoder** présente dans la classe principale *AppliSpringApplication* et son utilisation au sein de la méthode **.create()** de la classe *ServiceClientDirectImpl* .
- Analyser globalement tout le code de la partie **tp.appliSpring.security.generic.standalone** et la dépendance *jjwt-impl* dans pom.xml
- Compléter le code de la méthode **loadUserByUsername** de la classe **tp.appliSpring.bank.security.MyUserDetailsService**
- Analyser la configuration de la partie **restFilterChainWithoutOAuth2()** du fichier **tp.appliSpring.security.SecurityConfigForRest**
- Démarrer l'application SpringBoot avec les profils "**dev,reInit,withSecurity,withoutOAuth2**"
- Tester le comportement sécurisé de la partie rest via :
 - un **standaloneLogin/logout** pour bien repartir de zéro .
 - une première tentative de **compteAjax** avec un ajout de compte (label="ccc" , solde=0) menant normalement au message d'erreur 401/Unauthorized
 - une seconde tentative après un login en mode "**standaloneLogin (sans oauth2)**" via le compte "**client_1/pwd**"

6.5. Paramétrage @PreAuthorize sur CompteRestCtrl

- Ajouter sur certaines méthodes de la classe **CompteRestCtrl** des annotations **@PreAuthorize()** combinant si besoin `hasRole('ADMIN')` or `hasRole('CUSTOMER')` or `hasAuthority('SCOPE_resource.write')` de manière à ce que seuls les administrateurs , "client/customer" ou bien les personnes ayant les scopes suffisants (`SCOPE_resource.write` ou `SCOPE_resource.delete`) puisse déclencher les requêtes en mode POST,PUT et DELETE .
- En mode oAuth2 , tester le comportement affiné de la sécurité en tentant les modifications au sein de **compteAjax** après s'être connecté avec le compte "**user1/pwd1**" (**403/Forbidden**) puis "**admin1/pwd1**".
- En mode "standaloneLogin (sans oauth2)" , tester le comportement affiné de la sécurité en tentant les modifications au sein de **compteAjax** après s'être connecté avec le compte "**user1/pwd1**" (**403/Forbidden**) puis "**client_1/pwd**".

6.6. Sécurité sur partie site (thymeleaf) / Tp facultatif

- Analyser la configuration du fichier `tp.appliSpring.security.SecurityConfigForSite`
- On voit que le service `UserDetailsService` est également utilisé à ce niveau .
- Analyser les parties "login/logout" de la classe `tp.appliSpring.bank.site.controller.AppCtrl` ainsi que `advice/ErrorController`
- Analyser `src/main/resources/templates/login.html` et `error.html`
- Tester le comportement sécurisé de la partie "site" via :
 - "site" → "welcome" → "logout" pour bien repartir de zéro .
 - une navigation vers "espace_client" automatiquement reroutée vers "login" où il faut saisir "client_1/pwd" pour aller plus loin
 - un éventuel "logout" pour arrêter la session .

6.7. Test unitaire partiel facultatif de CompteRestCtrl en mode sécurisé

Coder (facultativement) un début de `TestCompteRestCtrlWithServiceMockWithOauth2Security` en s'inspirant de `TestHelloRestCtrlWithMocks.java.txt`

6.8. (facultatif) avec projet annexe "mysecurity-autoconfigure"

- Visualiser le contenu du projet annexe "mysecurity-autoconfigure" et la dépendance entre notre projet "debutApplispringWeb" et ce projet optionnel .
- Visualiser le déclenchement automatique de `xy.MySecurityConfig` via le fichier `org.springframework.boot.autoconfigure.AutoConfiguration.imports` présent dans la partie `src/main/resources` du projet `mysecurity-autoconfigure` (à construire et installer via "`mvn install`")
- Visualiser les paramétrages de type "`mysecurity.area.permit-all-list`" de `application-withSecurity.properties` qui sont spécifiquement analysés par le code interne de `mysecurity-autoconfigure`
- Faire évoluer le code de la classe `SecurityConfigForRest` en décommentant la partie "`//optional config from mysecurity-autoconfigure ... myPermissionConfigurer`" et en basculant de version de `restFilterChainBuilder()` .
- Déclencher "`mvn clean package`" sur le projet "debutAppliSpringWeb"
- Redémarrer l'application en mode sécurisé et tester le bon fonctionnement
- Mémoriser (via copies en commentaires) et modifier les valeurs de "`mysecurity.area.permit-all-list`" et "`mysecurity.area.permit-get-list`" de `application-withSecurity.properties`
- Redémarrer l'application en mode sécurisé et visualiser un comportement altéré
- Restaurer les valeurs appropriées de "`mysecurity.area.permit-all-list`" et "`mysecurity.area.permit-get-list`"
- Redémarrer l'application en mode sécurisé et tester le bon fonctionnement

7. Tp sur packaging et supervision d'appli spring

7.1. (tp facultatif) Déploiement d'une application Spring6/SpringBoot3 au format ".war" vers tomcat10

- Faire hériter la classe principale AppliSpringApplication de **SpringBootServletInitializer**
- Générer par assistant de l'IDE une redéfinition (@Override) de la méthode **configure()**
- placer ce code au sein de la méthode **configure()** :
return builder.sources(AppliSpringApplication.class).profiles("dev","reInit");
- ajouter temporairement <**packaging>war</packaging>** au sein de pom.xml
- construire **target/debutAppliSpringWeb.war** en lançant **mvn package** en mode **skipTest**
- installer tomcat10 en téléchargeant le .zip et en décompactant son contenu dans un répertoire de type c:\prog ou autre .
- Ajouter si besoin **set JAVA_HOME=chemin_menant_au_jdk17_ou21** au sein de c:\prog\tomcat10...\bin\startup.bat
- Démarrer tomcat via startup.bat
- Vérifier son bon fonctionnement via <http://localhost:8080>
- recopier debutAppliSpringWeb.war au sein du répertoire c:\prog\tomcat10...\webapps
- Vérifier le bon fonctionnement de l'application via <http://localhost:8080/debutAppliSpringWeb>
- arrêter tomcat10
- retirer ou mettre en commentaire <**packaging>war</packaging>** au sein de pom.xml

7.2. (tp facultatif) Déploiement d'une application Spring6/SpringBoot4 via docker

- Récupérer une copie du code de l'application sur une machine compatible docker (ex : linux ou bien windows + WSL2)
- construire une image docker en utilisant le fichier Docker file ou bien une cible maven adéquate
- lancer le conteneur docker à partir de l'image construite

7.3. (tp facultatif) actuator

- Activer certains "actuator" au sein de l'application et afficher certaines mesures via les URL REST adéquates