



NOVAO[®]
LEARNING

SPRING-BOOT :
LES BONNES
PRATIQUES

Pourquoi les Bonnes Pratiques ?



POURQUOI LES BONNES PRATIQUES ?

- **Maintenabilité** : Rendre le code plus facile à comprendre, modifier et entretenir.
- **Sécurité** : Réduire les risques de vulnérabilités et attaques en suivant des normes de sécurité préétablies. Validations de données, protections contre injections SQL, mis à jour des dépendances, etc.
- **Performance** : Optimise les performances de l'application en minimisant les multiples appels inutiles à des ressources, en minimisant la taille de certaines ressources, de la mise en cache, etc.
- **Compatibilité** : Permet de s'assurer de la bonne fonctionnalité de l'application dans différents environnements (client ou server).
- **Evolutivité** : Un code n'est jamais fixe mais devra sans cesse évoluer avec l'application. Les développeurs doivent mettre en place une structure et du code qui pense à cela et aux futurs autres développeurs qui devront continuer à travailler sur l'application.
- **Argent** : Tout cela amène à un gain de temps et donc un gain d'argent (une plutôt une non-perte d'argent dû à de la dette technique qui ralentira le développement ou maintenance d'une application).

ARCHITECTURES PACKAGES

INTRODUCTION

Une bonne organisation de package d'un projet est très important :

- Facilite la navigation
- Localisation des fonctionnalités
- Maintenabilité
- Plus simple à tester
- Gain de temps

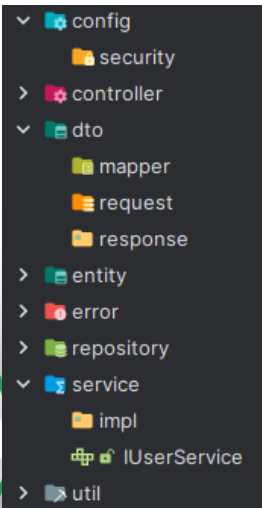
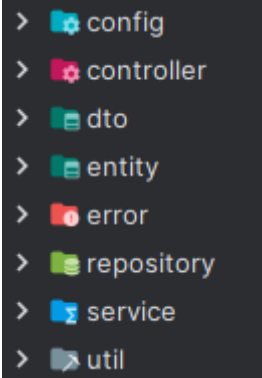
NOMENCLATURE

- Chaque package sur un projet JAVA doit être nommé en :
 - **lowercase**
 - Commencer par une lettre
 - Pas de caractères spéciaux
 - Pas de tirets (donc pas de kebab-case ou snake_case)
 - Si plusieurs mots, préférer créer plusieurs packages
 - Noms significatifs et simple
 - Noms standards (controller, service, etc.)
 - Normaliser les noms entre eux
 - Package principal nommé selon le nom de domain ou entreprise inverser (ex: com.insy2s)

DÉCOMPOSITION PAR COUCHES

La décomposition par couche sert à créer des packages qui feront chacun partie d'une couche de Spring. Pouvant contenir des sous-packages.

- **config** : contient tous les fichiers de configurations qui se lancera au run de l'application
- **controller** : contient tous les fichiers de la couche controller
- **dto** : contient les Data Transfert Object du projet (avec sous-packages si utile)
- **entity** (ou domain/model) : contient toutes les entités du projet
- **error** : contient tous les fichiers de gestions d'erreurs/exceptions
- **repository** : contient tous les fichiers de la couche repository
- **service** : contient tous les fichiers de la couche service (avec sous-package si besoin)
- **util** : contient tous les fichiers pour des fonctions utilitaires



DÉCOMPOSITION PAR FONCTIONNALITÉS

```
> auth
> cart
> order
> paiement
> product
```

La décomposition par fonctionnalités est le fait de créer des packages qui seront créés pour s'occuper de groupes de fonctionnalités. Contient donc tous les fichiers s'occupant de ces fonctionnalités quelque soit la couche.

Exemple:

- **auth** : est utilisé pour regrouper les fonctionnalités d'authentification et de gestion de compte
- **product** : est utilisé pour les fonctionnalités de gestion de produit
- **paiement** : est utilisé pour la gestion de paiement
- etc.

```
▼ auth
  AuthConfig
  AuthController
  Authority
  User
  UserController
  UserRepository
  UserService
> cart
> order
> paiement
> product
```

Popularisé par les frameworks FrontEnd, cette approche permet une approche logique des packages. Cela permet :

- Avoir des modules autonomes
- Une facilité de navigation
- Une meilleure gestion de la scalabilité

Separation of Concerns

INTRODUCTION

En français, Séparation des Préoccupations, est le fait de s'assurer que chaque classes et méthodes est censé avoir une utilité bien particulière, d'avoir une seule responsabilité.

Cela est fait pour aider à la lisibilité du code et à la recherche de fonctionnalités. Ainsi que s'assurer de la bonne application des règles de gestions du projet.

DIVISION LOGIQUE

Le principe consiste à identifier les différentes responsabilités ou préoccupation au sein d'un système et à les séparer en modules, composant ou couches distincts.

Chaque partie du système est alors responsable d'une tâche spécifique et a une préoccupation clairement définie.

RÉDUCTION DE LA COMPLEXITÉ

En séparant les préoccupations, la complexité globale du système est réduite, car chaque module ou composant peut être conçu implémenté et testé indépendamment des autres.

Cela permet également aux développeurs de se concentrer sur une tâche spécifique à la fois, ce qui facilite la compréhension du code et de la détection d'erreurs.

AMÉLIORATION DE LA RÉUTILISABILITÉ

En isolant les différentes préoccupations, les composants ou modules peuvent être réutilisés dans différents contextes sans nécessiter de modifications importantes.

Par exemple, un composant de gestion des utilisateurs peut être réutilisé dans différentes parties d'une application sans avoir à réécrire le code pour chaque utilisation.

FACILITATION DE LA MAINTENANCE

En isolant les préoccupations, les modifications ou les mises à jour dans une partie du système sont moins susceptibles d'affecter les autres parties.

Cela facilite la maintenance du code, car les modifications peuvent être apportées localement sans avoir à modifier de grandes parties du système.

CLARTÉ ET COHÉRENCE

En séparant les préoccupations, le code devient plus clair et cohérent, car chaque partie du système est responsable d'une tâche spécifique et suit des conventions de conception appropriées.

Lombok



INTRODUCTION

Dans un projet, avoir des plugins et librairies utilitaires est très utiles pour aider à faciliter le travail. Notamment en diminuant la répétition de code ou au renommage et correction de certaines fonctionnalités.

Parmi ces librairies, Lombok permet de réduire la quantité de code *boilerplate* (redondant et répétitif) en remplaçant ceci par des annotations qui les générera automatiquement.

Cela permet un code plus lisible, concis et facile à maintenir. Permet également de se concentrer davantage sur la logique métier de l'application plutôt que de créer des méthodes standard ou utilitaire.

INSTALLATION

```
<dependency>  
  <groupId>org.projectlombok</groupId>  
  <artifactId>lombok</artifactId>  
  <scope>provided</scope>  
</dependency>
```

Pour installer la librairie, il nous faut ajouter la dépendance suivante dans le fichier *pom.xml* du projet.

Il faut également ne pas oublier de recharger Maven.

FONCTIONNALITÉS

- `@Getter/@Setter` : Permet de générer automatiquement les getters et setters pour une classe ou attribut.
- `@ToString` : Génère automatiquement une méthode `toString()` dans la classe.
- `@NoArgsConstructor` / `@AllArgsConstructor` : Génère un constructeur vide ou un constructeur avec tout les attributs.
- `@RequiredArgsConstructor` : Génère un constructeur pour chaque attribut *final* de la classe.

EXEMPLES

```
@Entity
@Table(name = "users")
@Getter
@Setter
@ToString
public class User {

    @Id
    private Long id;
    private String firstName;
    private String lastName;

}
```

Créera les getters, les setters et le toString pour ces attributs.

```
@RestController
@RequiredArgsConstructor
public class UserController {

    private final UserService userService;
```

Créera un constructeur contenant UserService puisqu'il est en final.

DTO



INTRODUCTION

Les DTO (ou *Data Transfer Objects*) sont principalement utilisés pour transférer des données entre le client et le server (ou server/server dans le cas de multiple API ou architecture micro-services).

Il s'agit généralement de POJO, auquel on peut associer des fonctions de mapping pour faciliter leur création.

Cela sert à plusieurs choses:

- **Cacher la Conception** : permet de ne pas exposer la structure des données de notre database au client (et potentiellement personnes malveillantes). Moins ils en savent, plus on est sécurisé.
- **Protection de données sensibles** : évite de donner accès aux utilisateurs à des ressources potentiellement sensible. Ainsi que de donner potentiellement la possibilité de modifier des ressources qui ne devrait pas l'être . (Ex: « si je modifie mon nom/prénom je ne dois pas pouvoir modifier mon mot de passe également »).
- **Performance** : En donnant aux clients que ce dont il a besoin, on évite d'envoyer trop de ressources inutiles qui (en s'accumulant) pourrait ralentir l'application.
- **Compréhension** : Aide à la compréhension des données nécessaires. Notant lors de requêtes POST/PUT.

EXEMPLE

```
@Entity
@Table(name = "users")
public class User {

    @Id
    private Long id;
    private String username;
    private String password;
    private String email;
    private String firstName;
    private String lastName;
    private String role;
    private boolean enabled;
    private boolean accountNonExpired;
    private boolean credentialsNonExpired;
    private boolean accountNonLocked;
    private String profilePicture;
    private String coverPicture;
    private String resetToken;
    private String bio;
    @OneToOne
    private Address address;
}
```

```
public class AuthenticationRequest {

    private String username;
    private String password;
}
```

Pour se connecter, seulement besoin de deux champs.

```
public class UserSummary {

    private Long id;
    private String username;
    private String email;
    private String firstName;
    private String lastName;
    private String role;
    private String profilePicture;
}
```

Pour afficher une liste d'utilisateur, le client n'a besoin que d'une partie des données du user.

```
public class UserRegister {

    private String username;
    private String password;
    private String email;
    private String firstName;
    private String lastName;
    private String bio;
    private String road;
    private String zipCode;
    private String town;
}
```

Pour créer mon compte, je n'ai besoin que de certains champs spécifiques. Les autres seront créés automatiquement par le back.

MAPPER - MapStruct

INTRODUCTION

Dans une application backend, on devra souvent transformer certains objets en autre objet. Ceci est particulièrement le cas si on utilise des DTO.

Devoir faire à la main cette transformation à chaque fois peut poser quelques problèmes:

- Répétitivité de code
- Possible erreur de frappe / oublie de la part du développeur
- Maintenabilité complexe

Avoir une librairie de mapping peut être un gros plus. Comme *MapStruct*.

INSTALLATION

```
<dependency>
  <groupId>org.mapstruct</groupId>
  <artifactId>mapstruct</artifactId>
  <version>1.5.5.Final</version>
</dependency>

...

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.11.0</version>
  <configuration>
    <annotationProcessorPaths>
      <path>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct-processor</artifactId>
        <version>1.5.5.Final</version>
      </path>
      <path>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.30</version>
      </path>
      <path>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok-mapstruct-binding</artifactId>
        <version>0.2.0</version>
      </path>
    </annotationProcessorPaths>
  </configuration>
</plugin>
```

- Ajouter la dépendance dans le *pom.xml*
- On a également besoin d'un plugin
 - Si on utilise lombok, deux paths sont à rajouter dans les plugins pour la compatibilité de Lombok et MapStruct.

UTILISATION SIMPLE

```
@Mapper
public interface IUserMapper {
}
```

- Les mappers pour MapStruct sont des **interfaces**. Maven les implémentera automatiquement.
- `@Mapper` : spécifiera que cette interface utilise la dépendance *MapStruct*

```
@Mapper
public interface IUserMapper {
    UserSummary toUserSummary(User user);
}
```

Pour créer une fonction de mapping, il suffit de créer une fonction ayant en entrée l'objet qu'on a et en return ce qu'on veut.

Target methodName (*Source*)

Tant que les noms des champs sont les mêmes, tout sera automatisé.

UTILISATION EVOLUÉE

Si les noms ne correspondent pas entre la source et la target on peut le spécifier grâce à l'annotation `@Mapping`. Et cela même s'il s'agit d'objets relationnels pour lesquels on récupère juste les attributs.

MapStruct pourrait aussi être utilisé pour faire de l'algorithmie, comme par exemple récupérer juste la taille d'une liste. Grâce à l'option *expression*.

Dans le DTO on a `private String road`

Dans l'entité on a `private Address address`
Dans Address on a `private String road`

```
@Mapping(source = "road", target = "address.road")
@Mapping(source = "town", target = "address.city")
@Mapping(source = "zipCode", target = "address.zipCode")
User toUser(UserCreate userCreate);
```

IMPLÉMENTATION

Dans le dossier `/target/classes` qui est créé par Maven au run/installation du projet, on peut trouver les classes implémentant nos mappers.

Il est d'ailleurs conseillé de réinstallé le projet après création/modification des mappers pour être de leur bonne implémentation:

```
mvnw clean install
```

```
public UserSummary toUserSummary(User user) {  
    if (user == null) {  
        return null;  
    } else {  
        UserSummary userSummary = new UserSummary();  
        userSummary.setId(user.getId());  
        userSummary.setUsername(user.getUsername());  
        userSummary.setEmail(user.getEmail());  
        userSummary.setFirstName(user.getFirstName());  
        userSummary.setLastName(user.getLastName());  
        userSummary.setRole(user.getRole());  
        userSummary.setProfilePicture(user.getProfilePicture());  
        return userSummary;  
    }  
}
```

Non modifiable, ces classes permettent de vérifier ce que font MapStruct / Maven.

CONSTANTES

CONSTANTES GLOBALES

Par moment on aura des données fixes réutilisées plusieurs dans l'application:

- Rôle / Autorité des utilisateurs (ex: *admin*, *user*, *customer*, *manager*, etc.)
- Code erreur: notamment pour envoyer des erreurs au front
- Message: message d'erreur notamment.
- Constantes métiers: Spécifique au domaine métier de l'application. Ex: taux de tva, des codes pays, etc.

Ces données constantes ne doivent pas être utilisé tel quel dans le code, puisqu'en terme de compréhension ou maintenabilité cela devient très vite compliqué.

On préférera mettre ces données en constantes statiques soit dans le fichier où elles sont utilisées, soit dans des classes créées pour cela.

```
public class RolesConstant {  
  
    private RolesConstant() {  
    }  
  
    public static final String ADMIN = "ROLE_ADMIN";  
    public static final String USER = "ROLE_USER";  
    public static final String GUEST = "ROLE_GUEST";  
}
```

Il est aussi de bonnes pratiques de créer un constructeur privé dans des classes fait pour des données ou fonctions statiques (ou utilitaire) pour éviter toutes instantiations.

OBJET À DONNÉES CONSTANTES

RECORD

Introduit avec JAVA 14, les records sont des classes dont le seul but est de contenir des données immuables.

Cela aide à:

- Concis : réduit considérablement la quantité de code *boilerplate*
- Immuable : les attributs ne peuvent pas être modifiés après initialisation
- Lisibilité : le code étant concis il aide à la lisibilité en se focussant sur la donnée nécessaire
- Constructeur: un constructeur est créé automatiquement avec la création de la classe

```
public record AuthenticationRequest(  
    String username,  
    String password  
) {  
}
```

```
AuthenticationRequest authenticationRequest = new AuthenticationRequest("username", "password");
```


Couplage Faible

INSTALLATION

Le couplage faible est le fait que les différents composants du système sont interconnectés de manière minimale.

Cela signifie que les composants sont moins dépendants les uns des autres et peuvent fonctionner de manière autonome avec le moins d'interactions directes possible.

SERVICE

```
@RestController
@RequiredArgsConstructor
public class UserController {

    private final IUserService userService;

}
```

```
public interface IUserService {

    long countUsers();

}
```

```
@Service
@RequiredArgsConstructor
public class UserServiceImpl implements IUserService {

    private final IUserRepository userRepository;

    @Override
    public long countUsers() {
        return userRepository.count();
    }

}
```

Dans le cas de service, il s'agit de:

- Créer une interface contenant les traces des méthodes.
- Implémenter cette interface dans la classe de service
- Injecter la dépendance de l'interface dans le controller

Gestion d'erreur

INSTALLATION

La gestion d'erreur est très importante dans une application backend. Puisque la création de celle-ci n'est pas que l'implémentation de ce qui fonctionne mais aussi et surtout l'implémentation de ce qui ne doit pas fonctionner.

Pour cela on doit sans cesse vérifier des choses :

- Si un utilisateur est connecté
- Si un utilisateur est censé avoir accès à une ressource
- Si une ressource existe
- Si une donnée est conforme à des règles de gestions (nullable, unique, max ou min de caractères, etc.)
- Etc.

Et pour chacune de ces vérifications, on doit envoyer une erreur au client. Ce qui peut devenir très vite très compliqué en ayant sans cesse des conditions dans le code et des returns particuliers si on a des erreurs. Ce qui rend le code peu lisible et maintenable.

Pour nous aider à gérer cela, on utilise les exceptions.

CRÉER UNE EXCEPTION

Pour créer une exception, il suffit de créer une classe étendant de *RuntimeException*.

Dans le constructeur, on pourra simplement réutiliser le constructeur de *RuntimeException* pour redéfinir le message d'erreur.

```
public class UserNotFoundException extends RuntimeException {  
  
    public UserNotFoundException(String username) {  
        super("User not found with username: " + username);  
    }  
  
}
```

Pour l'utiliser il suffit de mettre *throw* avant l'utilisation du constructeur.
Throw étant une fonctionnalité qui, comme un *return*, arrête le processus.

```
@Override  
public User getByUsername(String username) {  
    Optional<User> user = userRepository.findByUsername(username);  
    if (user.isEmpty()) {  
        throw new UserNotFoundException(username);  
    }  
    return user.get();  
}
```

EXCEPTION HANDLER

```
@ControllerAdvice
public class ErrorsHandler {

}
```

Pour gérer les exceptions de manière globale, il nous faut d'abord créer une classe ayant l'annotation `@ControllerAdvice` qui permettra de créer des méthodes pour catcher des exceptions dans le run de notre application.

Pour catcher une exception bien particulière, il nous faut créer une méthode étant annoté avec `@ExceptionHandler`. On met la classe de l'exception dans les paramètres de l'annotation, et dans les paramètres de la fonction on pourra donc la récupérer.

Après de traiter celle-ci comme on veut, sachant que cette classe et ces fonctions permettent de retourner des réponses au client. Permettant de ne pas avoir à le faire dans les controllers spécifiques de notre backend.

```
@ExceptionHandler({UserNotFoundException.class})
public ResponseEntity<ErrorResponse> handleUserNotFoundException(UserNotFoundException e) {
    ErrorResponse error = new ErrorResponse(
        e.getMessage(),
        HttpStatus.NOT_FOUND.toString(),
        "NOT_FOUND",
        "USER_NOT_FOUND"
    );
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(error);
}
```

Validation des données

INSTALLATION

Même avec le traitement manuel des erreurs grâce aux exceptions handlers, celles-ci peuvent rester très laborieuses.

Si une entité a des attributs non nullable, avec des min ou max de caractères, des regex, des vérifications de dates futures ou passé, etc. Tout cela est à vérifier à chaque création et modification de cette entité et engendre de potentiellement problèmes:

- Répétitivité de code
- Perte de lisibilité
- Ralentissement de la maintenance
- Oublie de vérification

HIBERNATE VALIDATOR

Spring a une dépendance permettant d'automatiser ces vérifications pour envoyer directement une erreur au client si elles ne relèvent une erreur.

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

Cette dépendance permet d'avoir accès à tout un panel d'annotation permettant de spécifier des restrictions à différents types d'attributs et d'enclencher le check automatique de celles-ci à volonté.

PREMIÈRES VALIDATIONS

```
@NotBlank
@Size(min = 3, max = 20)
private String username;

@NotNull
@Past
private LocalDate birthDate;

@NotBlank
@email
private String email;

@NotBlank
@Pattern(regexp = "^(?=.*[A-Za-z])(?=.*\\d)[A-Za-z\\d]{8,}$")
private String password;
```

- **@NotNull** : précise qu'une valeur ne peut pas être null
- **@NotBlank** : précise qu'un string ne peut être null ou vide ou ne contenir que des espaces
- **@Size** : précise qu'une donnée doit être supérieur et/ou inférieur au(x) min/max spécifié(s)
- **@Past** : précise qu'une date est strictement dans le passé
- **@Email** : précise que le string doit avoir un format d'email
- **@Pattern** : précise que le string a un regex particulier

VALIDER

```
@PostMapping("/users")
public ResponseEntity<Long> createUser(@RequestBody @Valid UserCreate user) {
    Long userId = userService.create(user);
    return ResponseEntity.status(HttpStatus.CREATED).body(userId);
}
```

Spring Validation permet grâce à l'annotation `@Valid` d'enclencher automatiquement la validation de la donnée suivie avant le déroulement de l'exécution du controller.

Si une erreur est détectée, une exception *MethodArgumentNotValidException* est automatiquement *throw* puis catché pour envoyé une réponse BAD REQUEST au client.

Si on utilise des DTO, il faut évidemment que ceux-ci contiennent également des validateurs.

EXCEPTION HANDLER

Etant donné que l'annotation *throw* une exception, on peut nous-même la catcher grâce à un exception handler pour customiser la réponse que l'on veut envoyer au client.

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(MethodArgumentNotValidException.class)
public Map<String, String> handleValidationExceptions(MethodArgumentNotValidException ex) {
    Map<String, String> errors = new HashMap<>();
    ex.getBindingResult().getAllErrors().forEach((error) -> {
        String fieldName = ((FieldError) error).getField();
        String errorMessage = error.getDefaultMessage();
        errors.put(fieldName, errorMessage);
    });
    return errors;
}
```

MESSAGES CUSTOMISÉS

On a par défaut des messages d'erreur prédéfinis par la librairie qui sont en anglais. On peut néanmoins les redéfinir. Soit:

- Directement dans les annotations:

```
@NotBlank(message = "Le nom d'utilisateur est obligatoire")
@Size(min = 3, max = 20, message = "Le nom d'utilisateur doit être entre 3 et 20 caractères")
```

- Dans un fichier de configuration *ValidationMessages.properties* créé dans le dossier *resources*:

```
jakarta.validation.constraints.NotBlank.message=ne doit pas être vide.
jakarta.validation.constraints.NotEmpty.message=ne doit pas être vide.
jakarta.validation.constraints.NotNull.message=ne doit pas être null.
jakarta.validation.constraints.Null.message=doit être null.
jakarta.validation.constraints.Email.message=doit être une adresse email bien formée
jakarta.validation.constraints.Pattern.message=doit correspondre à "{regex}"
jakarta.validation.constraints.Size.message=la taille doit être comprise entre {min} et {max}
jakarta.validation.constraints.AssertTrue.message=doit être vrai
jakarta.validation.constraints.AssertFalse.message=doit être faux
jakarta.validation.constraints.DecimalMax.message=doit être inférieur ou égal à {value}
jakarta.validation.constraints.DecimalMin.message=doit être supérieur ou égal à {value}
jakarta.validation.constraints.Digits.message=nombre de chiffres significatifs doit être inférieur ou égal à {integer} et nombre de chiffres après la virgule doit être inférieur ou égal à {fraction}
jakarta.validation.constraints.Future.message=doit être une date dans le futur
jakarta.validation.constraints.FutureOrPresent.message=doit être une date dans le présent ou dans le futur
jakarta.validation.constraints.Past.message=doit être une date dans le passé
jakarta.validation.constraints.PastOrPresent.message=doit être une date dans le présent ou dans le passé
jakarta.validation.constraints.Max.message=doit être inférieur ou égal à {value}
jakarta.validation.constraints.Min.message=doit être supérieur ou égal à {value}
jakarta.validation.constraints.Negative.message=doit être négatif
jakarta.validation.constraints.NegativeOrZero.message=doit être négatif ou égal à zéro
jakarta.validation.constraints.Positive.message=doit être positif
jakarta.validation.constraints.PositiveOrZero.message=doit être positif ou égal à zéro
```

AUTRE

Documentation: <https://beanvalidation.org/2.0/spec/#builtinconstraints>

Les validateurs fournis par la librairie sont nombreux et permettent d'avoir des vérifications avancées sur nos données. Mais également, elle nous permet de créer nous-même nos validateurs pour par exemple:

- Avoir un validateur de mot de passe plutôt que d'utiliser des regex
- Avoir un validateur pour du fichier et définir par exemple le type MIME possible

Profiles



INTRODUCTION

Une application est censé run dans différents types d'environnement:

- Environnement de Développement
- Environnement de Production
- Environnement de Test
- Environnement de Préproduction

Chacun de ces environnements peuvent avoir des configurations différentes:

- Mot de passes
- URL (BDD, front, etc.)
- Secret Key
- Niveau de log
- Etc.

On a donc besoin de pouvoir facilement définir tout cela dans des fichiers de configurations différents pour chacun de ces environnements.

POM.XML

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
  .....
</build>

<profiles>
  <profile>
    <id>dev</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
      <spring.profiles.active>dev</spring.profiles.active>
    </properties>
  </profile>
  <profile>
    <id>prod</id>
    <properties>
      <spring.profiles.active>prod</spring.profiles.active>
    </properties>
  </profile>
</profiles>
```

On doit rajouter plusieurs choses dans le pom.xml :

- Profiles : pour créer les différents profiles de l'application
- Resource : pour spécifier à spring où sont les fichiers properties des profiles

FICHIERS PROPERTIES

On aura donc plusieurs fichiers properties:

- *application.yml* : propriétés par défaut de l'application
- *application-dev.yml* : propriétés du profile dev
- *application-prod.yml* : propriétés du profile prod
(j'ai rajouté des variables d'environnement système pour simuler qu'on est sur un serveur avec des configurations locales qui doivent être cachés)

```
spring:
  application:
    name: first-app-spring
  profiles:
    default: dev
    active: @spring.profiles.active@
```

```
server:
  port: 8181
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/databasetest
    username: postgres
    password: postgres
  jpa:
    hibernate:
      ddl-auto: create-drop
    properties:
      hibernate:
        show_sql: true
        format_sql: true
```

```
server:
  port: 9753
spring:
  datasource:
    url: jdbc:postgresql://${DB_HOST}:${DB_PORT}/${DB_NAME}
    username: ${DB_USER}
    password: ${DB_PASS}
  jpa:
    hibernate:
      ddl-auto: none
```

RUN

- Lancer le profile par défaut: *dev*

```
mvnw spring-boot:run
```

- Lancer un profile particulier: *prod*

```
mvnw spring-boot:run -Pprod
```

Documentation



INTRODUCTION

Dans une application, il est important de documenter pour:

- Mise en place de l'environnement
- Description des fonctionnalités
- Descriptions des fichiers
- Explications métier
- Loguer les événements du run de l'application

README

Un fichier *README.md* à la racine du projet est très utile et **nécessaire** pour plusieurs raisons:

- Description du projet
- Installation de l'environnement
- Dépendances nécessaires (comme avoir une BDD, de quel type, nom, credential)
- Run de l'application
- Secondaire:
 - Contact
 - Description des fonctionnalités
 - Images
 - Lien vers documentations externes

DOCUMENTATION DE CLASSES ET MÉTHODES

- Les documentations de classes permettent de savoir à quoi servent celle-ci, si elles sont reliées à d'autres, et également d'en décrire la notion métier.

```
/**  
 * Entité User  
 * Permet de définir les attributs d'un utilisateur et de les lier à la base de données.  
 * Il s'agit d'avoir les informations permettant de traiter les comptes utilisateur  
 * pour qu'ils puissent se connecter, interagir avec l'application et accéder à leurs informations personnelles.  
 */
```

- Les documentations de méthodes permettent de savoir ce qu'elles font, possiblement les restrictions, la description métier, ce qu'on attend en paramètre, et les différents return/throw possibles

```
/**  
 * POST /users : Permet de créer un nouvel utilisateur.  
 * Ce endpoint est destiné uniquement à l'administrateur de l'application,  
 * puisqu'il permet de créer un nouvel utilisateur avec un rôle spécifique,  
 * ce qu'un utilisateur lambda ne peut pas faire.  
 *  
 * @param user UserCreate : les informations de l'utilisateur à créer  
 * @return Status 201 - Created : l'identifiant de l'utilisateur créé  
 * Status 400 - Bad Request : si les informations de l'utilisateur ne sont pas valides  
 */
```


COMMENTAIRES

Si en apprentissage les commentaires dans le code peuvent être très utiles pour garder en mémoire le fonctionnement de celui-ci, pourquoi on le fait, à quoi sert une ligne ou bloc de code, en entreprise on évite un maximum de mettre des commentaires dans le code.

La règle étant: « *Si on a besoin de décrire notre code, c'est que notre code est mauvais* ».

La documentation, le nom et le code en lui-même doivent être explicite et compréhensible. Les commentaires pouvant par moment alourdir la lecture du code à trop être descriptif de quelque chose déjà limpide.

L'exception à cette règle est quand il y a une forte composante métier dans un bloc de code dont il serait nécessaire d'être explicite.

LOG

INTRODUCTION

Les logs sont des enregistrements générés par l'application pour capturer des informations sur son fonctionnement pendant son run en temps réel.

Ils sont utiles pour permettre de savoir ce qu'il se passe dans une application pendant qu'on l'utilise (que ce soit en dev, debugging ou en prod), ainsi que de garder une trace de ce fonctionnement sur la durée.

Ils peuvent être enregistrés dans des fichiers qui pourront être archivés ou générés au crash de l'application.



LOG

NIVEAUX

Principaux niveaux de logs:

- **Debug** : Utilisé pour le débogage permettant en environnement de dev de décrire de manière détaillée sur le fonctionnement. On ne les affiche généralement pas en production.
- **Info** : Utilisé pour décrire des messages informatifs, pouvant être non technique, pour décrire les événements important de l'application.
- **Warning** : Utilisé pour décrire des avertissements pour des situations potentiellement problématique (une donnée incorrecte, un mauvais accès à une ressource, etc.), mais pas grave.
- **Error** : Utilisé pour décrire des erreurs ou exceptions dans le processus de l'application. Généralement utilisé pour mettre des flags pour qu'une personne puisse traiter cette erreur.

LOG USAGE

```
private Logger log = Logger.getLogger(UserController.class.getName());

@GetMapping("/users")
public List<UserSummary> getAll() {
    log.info("REST request to get all users");
    return userService.findAll();
}
```

Pour utiliser les logs il faut instancier *Logger* dans la classe où on veut l'utiliser, qu'il faudra paramètre avec le nom de la classe.

Puis il faudra juste utiliser `log.loglevel(«mon message »)` ; pour créer un log au niveau choisi

```
@Slf4j
public class UserController {
```

En utilisant lombok, pour instancier *Logger* il suffit d'annoter la classe avec `@Slf4j`

```
2024-02-14T13:53:06.467+01:00 INFO 30528 --- [first-app-spring] [nio-8181-exec-1] c.i.f.controller.UserController : REST request to get all users
```

LOG

PROPERTIES

Par défaut, seuls les logs de niveau **INFO** et au-dessus (**WARN** et **ERROR**) sont activé et visible dans la console. Pour voir ceux d'un niveau plus bas, il faudra configurer cela dans les properties du projet.

```
logging:  
  level:  
    com.insy2s: DEBUG
```

Précise pour quel package on veut le niveau de log minimal. Ici le package principal du projet est *com.insy2s* donc on met ça pour avoir le niveau de log en **DEBUG** pour le projet.

LOG FILE

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

  <property name="LOGS" value="./logs"/>

  <appender name="Console"
    class="ch.qos.logback.core.ConsoleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>
        %white(%d{ISO8601}) %highlight(%-5level) [%blue(%t)] %yellow(%C): %msg%n%throwable
      </Pattern>
    </layout>
  </appender>

  <appender name="RollingFile" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>${LOGS}/project-logger.log</file>
    <encoder
      class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
      <Pattern>%d %p %C [%t] %m%n</Pattern>
    </encoder>

    <rollingPolicy
      class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <!-- rollover daily and when the file reaches 10 MegaBytes -->
      <fileNamePattern>${LOGS}/archived/project-logger-%d{yyyy-MM-dd}.%i.log
      </fileNamePattern>
      <timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
        <maxFileSize>10MB</maxFileSize>
      </timeBasedFileNamingAndTriggeringPolicy>
    </rollingPolicy>
  </appender>

  <root level="info">
    <appender-ref ref="RollingFile"/>
    <appender-ref ref="Console"/>
  </root>

  <logger name="com.insy2s" level="trace" additivity="false">
    <appender-ref ref="RollingFile"/>
    <appender-ref ref="Console"/>
  </logger>

</configuration>
```

En créant un fichier dans le dossier *resources*, nommé *logback-spring.xml*, on peut configurer les logs pour notamment:

- Modifier les formats de log
- Colorer les logs
- Enregistrer les logs dans un fichier
- Archiver automatiquement les fichiers dès qu'ils atteignent une taille particulière.

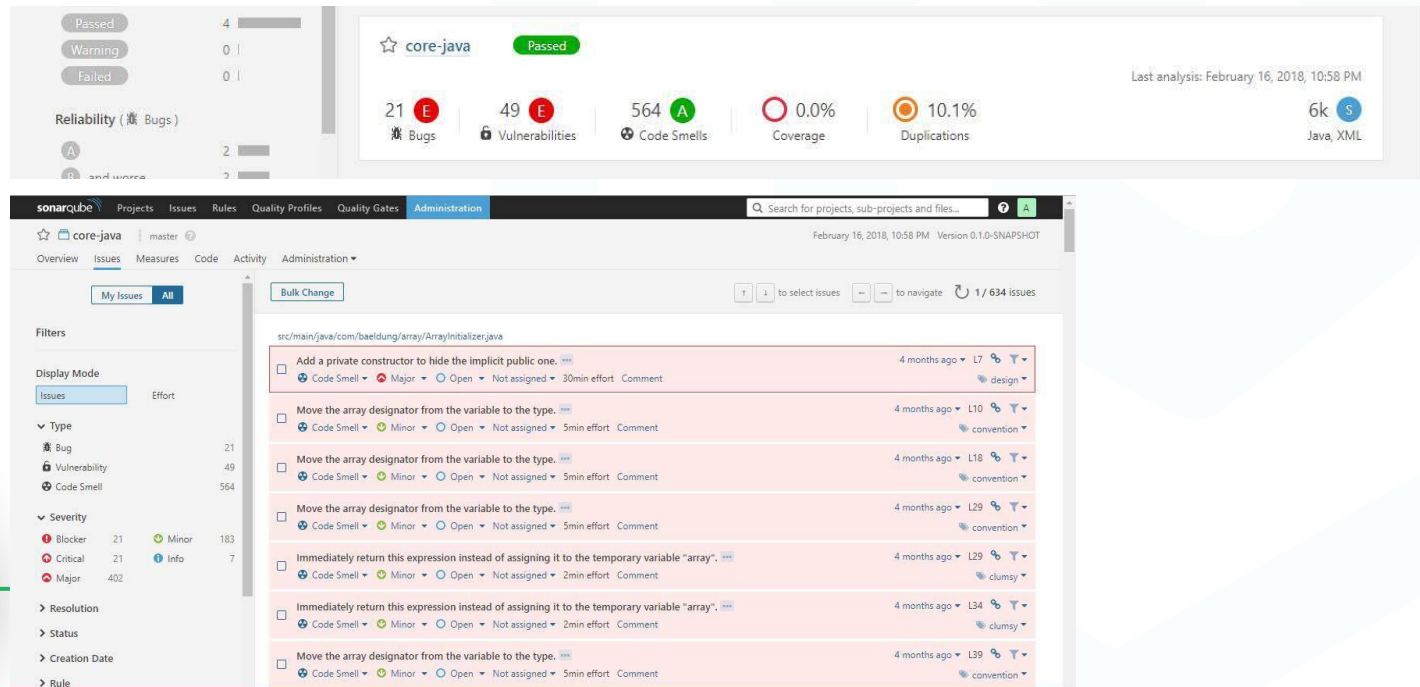
SonarQube — SonarLint

SONARQUBE

SonarQube est une plateforme OpenSource conçue pour évaluer et améliorer la qualité du code sources des projets logiciels (pas seulement Spring).

Elle offre une gamme de fonctionnalités pour l'analyse statique du code, l'identification des problèmes de qualité, la mesure de la dette technique et la gestion de la qualité du code.

Cette plateforme peut être lancée sur un serveur (local ou distant) pour scanner le code. Particulièrement utile pour générer des rapports de qualité ainsi que de lancer des checks pour approuver un lancement de l'application en production.

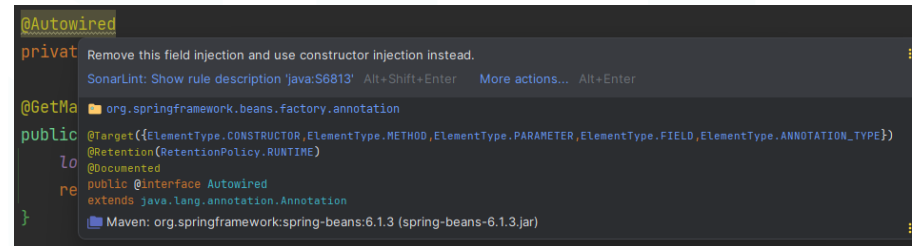
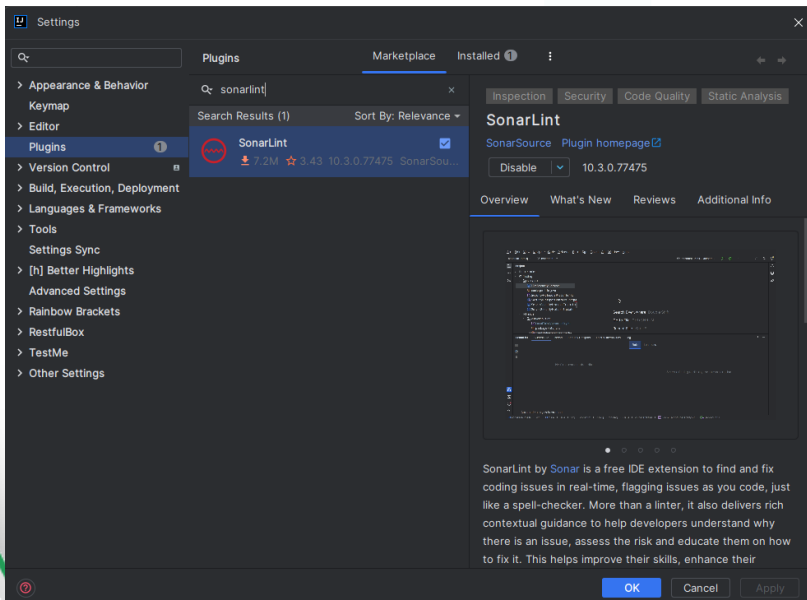


The screenshot displays the SonarQube web interface for a project named 'core-java'. The top section shows overall quality metrics: 21 Bugs, 49 Vulnerabilities, 564 Code Smells, 0.0% Coverage, and 10.1% Duplications. The last analysis was performed on February 16, 2018, at 10:58 PM. Below this, the 'Issues' tab is selected, showing a list of issues with details such as severity, effort, and resolution status. The left sidebar contains filters for display mode (Issues, Effort), type (Bug, Vulnerability, Code Smell), severity (Blocker, Critical, Major, Minor, Info), resolution, status, creation date, and rule.

SONARLINT

Puisque devoir lancer un server en local pour scanner le code peut prendre du temps (pour l'installation mais aussi pour chaque scan), Sonar a créé une extension IDE appelée: **SonarLint**. Il est fortement conseillé de l'utiliser.

Sur IntelliJ, en allant dans: File > Settings > Plugins > Marketplace, on peut rechercher le plugin et l'installer.



IntelliJ affichera les warnings directement dans le code en temps réel. On pourra donc ouvrir un descriptif du warning pour en voir:

- Le niveau de critique
- Pourquoi ce warning
- Comment le corriger

Transaction

INTRODUCTION

Une transaction est une unité logique d'opération qui doit être exécutée de manière atomique, c'est-à-dire soit toutes les opérations réussissent et sont validées, soit aucune opération n'est acceptée (*rollback*).

Les transactions sont couramment utilisées pour garantir la cohérence des données dans les applications en cas d'opérations multiples qui doivent être exécutées de manière cohérente.

Exemple:

1. Lors de la création de compte, je dois enregistrer une entité User et une entité Address.
2. User ayant la relation avec Address, j'enregistre d'abord l'adresse avant d'enregistrer les données de l'utilisateur.
3. Si la création de l'adresse fonctionne mais que pour une quelconque raison la création de l'utilisateur ne fonctionne pas, il faudrait revenir en arrière et annuler la création de l'adresse.

UTILISATION

On utilise simplement l'annotation : `@Transactional` sur une méthode, ou directement sur une classe (le plus souvent les services) pour la donner automatiquement à toutes ses méthodes.

```
@Service
@RequiredArgsConstructor
@Transactional
public class UserServiceImpl implements IUserService {
```

⚠ **Important** ⚠ Transactional doit bien être importé de [org.springframework.transaction.annotation.Transactional](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.transaction.annotation.Transactional.html)

READONLY

L'annotation Transactional a des options, l'une d'entre elle très utile est *readOnly*.

Celle-ci permet plusieurs choses:

- On assure que la transaction est readOnly, donc qu'on ne fait que de la récupération, aucune modification n'est pas possible.
- Pour certaines BDD (tel que PostgreSQL) améliore la performance des requêtes.

```
@Override  
@Transactional(readOnly = true)  
public User getByUsername(String username) {
```

Early Exit



EARLY EXIT

Le principe de l'Early Exit est le fait que dans un bloc if/else, si l'un d'entre eux est là pour arrêter le processus et l'autre pour le continuer, on préfère mettre celui qui arrête l'exécution en premier.

Cela permet une meilleure lisibilité de code, ce qui aide grandement à la maintenance, débbugging et évolution du code.



```
if (condition) {  
    // bloc d'exécution si la condition est vraie  
} else {  
    // bloc d'exécution si la condition est fausse  
    return; // ou une autre instruction qui arrête l'exécution  
}  
// Code à exécuter après le if/else  
  
.....
```



```
if (!condition) {  
    // bloc d'exécution si la condition est vrai  
    return; // ou une autre instruction qui arrête l'exécution  
}  
// exécution si la condition est fausse
```