Doc technique Candy

|  |  |
| --- | --- |
| Rédigé par | Poste |
| Marc Flausino | Chef de projet technique |

Table des matières

[Eléments techniques 1](#_Toc139875886)

[Projet 1](#_Toc139875887)

[Dépendances Maven 2](#_Toc139875888)

[Base de données 3](#_Toc139875889)

[API 3](#_Toc139875890)

[Bonnes pratiques 4](#_Toc139875891)

[Tests 4](#_Toc139875892)

[Test des routes 4](#_Toc139875893)

[Tests JUnit 5](#_Toc139875894)

[Aide 5](#_Toc139875895)

[Spring 5](#_Toc139875896)

[Base de données 5](#_Toc139875897)

[Hibernate/JPA 5](#_Toc139875898)

[Nommage des tables et colonnes en base 6](#_Toc139875899)

[Repository 6](#_Toc139875900)

[Jackson 6](#_Toc139875901)

[Tests 7](#_Toc139875902)

# Eléments techniques

Voici mes consignes pour la création du POC suite au cahier des charges reçu de Mme Dragibus.

## Projet

Vous devrez créer un projet java avec [**Spring Initializer**](https://start.spring.io/) selon les critères suivants :

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

***A noter :*** *on package en « war » par habitude pour éventuellement le déployer sur un serveur Tomcat, mais dans le cadre du projet vous pouvez lancer le projet en mode « standalone ».*

### Dépendances Maven

Vous ajouterez les dépendances suivantes pour ajouter la librairie JPA et le client de connexion mysql :

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>  
<dependency>  
 <groupId>com.mysql</groupId>  
 <artifactId>mysql-connector-j</artifactId>  
</dependency>

*[Optionnel]*Vous ajouterez la dépendance lombok pour les annotations assez pratiques 🡪 @Getter @Setter @Builder @Data :

<dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>1.18.26</version>  
</dependency>

A noter : Si vous ajoutez lombok, il faut effectuer [quelques configurations sur votre IDE](https://www.baeldung.com/lombok-ide).

Si besoin, vous pourrez ajouter d’autres dépendances au projet.

## Base de données

Vous travaillerez avec une base de données Mysql (MariaDB accepté).

Il vous est demandé de créer le **MCD** et le **MPD** de la base et vous devrez **impérativement** les faire valider par votre formateur avant de vous lancer sur l’API.

Tous ces éléments seront stockés en base :

* Les utilisateurs
  + En plus de la pkey, il ne faut qu’un seul champ pour le nom d’utilisateur, ça suffira pour ce POC
* Les commandes avec le détail de chacune de ses CandyBox., et bien sûr le candytag concerné par la commande !
* Les couleurs
* Les candytag
  + A noter que dans le code vous utiliserez une *enum* pour gérer les candytag (cf *@Enumerated(EnumType.STRING)*)

## API

Vous devrez créer une API qui utilise JPA/Hibernate pour l’accès en base et la modélisation des objets métiers.

Cette API contiendra une route pour permettre d’effectuer une commande :

Route avec méthode POST : /api/candy/order

Elle consomme un JSON qui contient :

|  |  |  |  |
| --- | --- | --- | --- |
| Propriété | Type | Exemple de valeur | Définition |
| candytag | string | "ASAGAO" | CandyTag |
| qte | number | 65 | Quantité |
| idusers | number | 123 | Id de l’utilisateur |

Elle produit en réponse un JSON qui contient :

|  |  |  |  |
| --- | --- | --- | --- |
| Propriété | Type | Exemple de valeur | Définition |
| order | objet | Voir exemple plus bas | Objet représentant un order |
| msgerr | string | Chaîne vide : "" | Valorisé s’il y a une erreur |
| status | number | 200 | Code numérique représentant le status ok (200) ou en erreur (400) |
|  |  |  |  |

Exemple :

{

    "order": {

        "idorder": 1,

        "candytag": {

            "idcandytag": 2,

            "candytag": "MOMO"

        },

        "candybox": [

            {

                "idcandybox": 1,

                "itemcandybox": [

                    {

                        "iditemcandybox": 1,

                        "qte": 1,

                        "color": {

                            "idcolor": 2,

                            "color": "bleu"

                        }

                    },

                    {

                        "iditemcandybox": 2,

                        "qte": 2,

                        "color": {

                            "idcolor": 3,

                            "color": "rouge"

                        }

                    }

                ]

            }

        ]

    },

    "msgerr": "",

    "status": 200

}

## Bonnes pratiques

Il vous est demandé de travailler en suivant les bonnes pratiques :

* Pas de code métier dans les controlleurs, vous utiliserez des Services Spring (cf annotation @Service) et éventuellement des classes dédiées pour les algorithmes (cf ci-dessous).
* On évitera également d’effectuer les accès en base depuis le controlleur, là encore on devra utiliser des Repository via les services.
* Vous commenterez votre code avec du JavaDoc, les variables ont des noms cohérents et vous factorisez au mieux votre code.

Pour les différentes règles de gestion liées à chaque candytag, il est fortement conseillé de créer des classes *processor* à part. Par exemple, une classe KikuProcessor chargée d’effectuer l’algo lié à Kiku. La méthode du processor qui sera appelée devra renvoyer un objet représentant la commande que l’on veut sauvegarder en base. Le traitement qui recevra cet objet sera en charge de faire la sauvegarde en base de toute la commande (avec son contenu).

Si vous procédez de cette façon vous pourrez tester plus facilement vos algos sans qu’il y ait eu d’écriture en base.

## Tests

### Test des routes

Vous testerez votre route avec POSTMAN en mettant en place plusieurs tests différents et en utilisant à chaque fois l’onglet **Tests** pour vérifier la réponse.

// test retour

pm.**test**("Test si retour ok", () **=>** {

    pm.response.to.have.status(200);

});

//test msgerr vide

pm.**test**("Test msgerr vide", () **=>** {

    let responseData **=** pm.response.json();

    pm.expect(responseData.msgerr).to.eql("");

});

Vous devrez également tester les cas avec réponse d’erreur.

### Tests JUnit

Vu la complexité des règles de gestion par CandyTag, vous devrez créer les tests unitaires pour chaque algo correspondant aux CandyTag (chaque CandyTag développé).

Pour tester chaque algo, il faudra penser à tester :

* Les cas optimistes, tout est normal
* Les cas pessimistes, on ne peut pas effectuer la commande
  + *Par exemple, je veux faire une commande KIKU de 17 bonbons, 17 n’est pas divisible par 6 donc la commande doit être refusée*

# Aide

## Spring

Attention, vous devez penser en tant que composants Spring si vous voulez accéder aux différents « bean » instanciés par Spring. En d’autres termes, si vous souhaitez utiliser des classes repository dans une classe il faut que cette dernière soit un « bean » également (via annotations @Service, @Component, etc.).

## Base de données

Si vous le souhaitez et si vous avez les *skills*, vous pouvez générer la base de données directement depuis votre modèle dans MYSQL Workbench 🡪 Database / Forward engineer

## Hibernate/JPA

Doc : <https://blog.paumard.org/cours/jpa/chap03-entite-entite.html>

Si vous le souhaitez et si vous avez les *skills*, vous pouvez générer les entités à partir de la base de données dans IntelliJ ou Eclipse. **ChatGPT** sait aussi générer les entités avec les requêtes DDL. Il faut bien lui donner le contexte cependant (langage, techno, etc.).

Attention, si votre table de commande s’appelle **order**, c’est un mot réservé du langage sql, il faudra mettre le nom de table entre guillemets dans l’entité :

@Table(name = "\"Order\"", schema = "candy", catalog = "")

**Pour éviter tout problème, il est préférable d’éviter les noms réservés du langage SQL !**

Annotations utiles :

@ManyToOne, @OneToMany : <https://www.baeldung.com/hibernate-one-to-many>

@GeneratedValue : <https://www.baeldung.com/hibernate-identifiers>

@Enumerated(EnumType.*STRING*) : <https://www.baeldung.com/jpa-persisting-enums-in-jpa>

### Nommage des tables et colonnes en base

**ATTENTION** : Si vous avez un MAC et des majuscules dans vos noms de table ou colonnes en base de données, ajoutez cette configuration dans le fichier **properties** pour que Spring JPA ne renomme pas implicitement vos entités…

spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl

## Repository

Vous devez déclarer une interface de type CrudRepository qui vous permettra de faire des requêtes et d’agir sur la base de données.

Par exemple pour déclarer une interface qu’on appellera IColorRepository en charge du model ColorEntity.

public interface IColorRepository extends CrudRepository<ColorEntity,Long > {}

De base, l’interface aura toutes les commandes de base CRUD mais si on veut ajouter une requête particulière, cela est possible en ajoutant dans l’interface la signature de méthode souhaitée avec l’annotation @Query.

Exemple avec [possibilité de trier via Sort](https://www.baeldung.com/spring-data-sorting) :

*/\*\*  
 \* Permet de recuperer toutes les couleurs via JPQL avec un parametre Sort  
 \* pour nous donner la possibilite de trier les couleurs  
 \* @param sort : objet Sort  
 \* @return la liste de couleurs  
 \*/*@Query("SELECT c FROM ColorEntity c")  
Collection<ColorEntity> getColors(Sort sort);

***Attention*** : les requêtes sont du code [JPQL](https://www.labri.fr/perso/rgiot/cours/jee/jpa/JPQL.pdf), ce n’est pas du langage SQL même si ça y ressemble. C’est comme si vous faisiez du SQL qui requête **vos entités** !

Pour récupérer une instance de notre interface là où on en a besoin, il faudra le faire via un @Autowired ou dans un [constructeur implicitement](https://www.baeldung.com/constructor-injection-in-spring#implicit-constructor-injection) (depuis version 4.3 de Spring).

## Jackson

Annotations utiles :

@JsonIgnore : <https://www.baeldung.com/jackson-annotations#bd-2-jsonignore>

## Tests

<https://www.baeldung.com/spring-boot-testing>

Annotations utiles :

@SpringBootTest

@Autowired

Voici un exemple de fragment de classe pour montrer un exemple d’initialisation de classe de test d’un processor :

@SpringBootTest  
class KikuProcessorTest {  
  
 @Autowired  
 ProcessorFactory processorFactory;

IProcessor processor;  
  
 @BeforeEach  
 void setUp() {  
 processor = processorFactory.getProcessor(CandyTag.*KIKU*);  
 }

Dans cet exemple, on utilise une classe **Factory**. C’est une classe **qu’on a créé** et qui est chargée de me donner une instance d’un objet. C’est une pratique courante en développement. Une autre pratique courante est de manipuler des interfaces dans nos méthodes. On aurait pu renvoyer la classe *KikuProcessor* mais on renvoie l’interface qu’elle implémente : *IProcessor*. De cette façon, notre code est réduit car tous mes processors implémentent cette même interface. Si je n’avais pas fait cela, j’aurais eu autant de lignes que de processors.