

Guide du développeur et document  
d'exploitation pour l'adoption de  
l'approche API First par l'équipe Squad  
Digital.



## Table des matieres

<b>INTRODUCTION .....</b>	<b>3</b>
<b>L'APPROCHE API FIRST ET LES OUTILS OPENAPI .....</b>	<b>3</b>
LES OUTILS OPENAPI.....	4
<i>Générez la structure de code initiale en utilisant la fonction de         génération de code des outils OpenAPI. ....</i>	<i>5</i>
RÉSUMÉ .....	7
<b>APPROCHE API-FIRST, ARCHITECTURE LABEL-FACTORY .....</b>	<b>8</b>
API-METIER-LBV FOURNISSEUR SWAGGER .....	9
<i>Note ⚠ : .....</i>	<i>10</i>
BACKEND GENERATOR CI/CD.....	11
FONDEMENT DU GÉNÉRATEUR - .GITLAB-CI.YML : .....	11
API_X REPOSITORY .....	14
RÉSUMÉ : .....	15
<b>COMMENCER ET INSTALLATION.....</b>	<b>16</b>
<i>Nexus Configuration:.....</i>	<i>16</i>
<i>Générée Target.....</i>	<i>20</i>
RÉSUMÉ:.....	21
<b>L'IMPLÉMENTATION DANS SPRING BOOT .....</b>	<b>22</b>
<i>Modèle délégué dans le code généré par Swagger? .....</i>	<i>24</i>
TESTS DE L'IMPLÉMENTATION DU CODE SUR L'APPLICATION SPRING AVEC POSTMAN : .....	26
<b>CONCLUSION: .....</b>	<b>27</b>

# Introduction

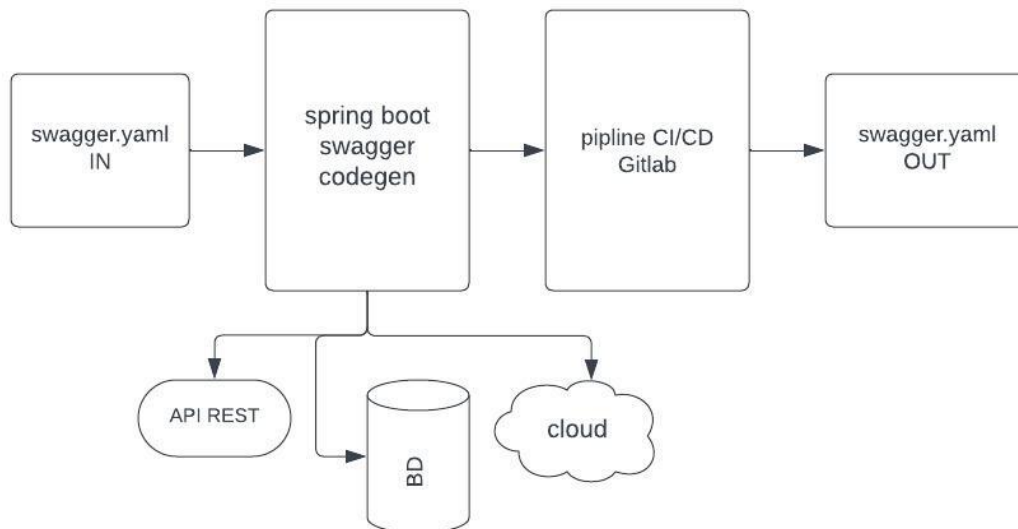
"Design First" est une approche du développement d'API où la conception de l'API est priorisée avant l'écriture de tout code. Cette approche implique la création d'une spécification détaillée de l'API à l'aide d'outils tels que Swagger et OpenAPI, qui peut ensuite être partagée avec les autres membres de l'équipe et les parties prenantes. En se concentrant d'abord sur la conception de l'API, les équipes peuvent garantir que l'API est bien structurée et répond aux besoins des utilisateurs avant d'écrire le moindre code. Cela peut permettre de gagner du temps et des ressources à long terme en évitant la nécessité de modifications ou de refontes importantes ultérieurement. L'architecture de l'API désigne la structure générale et la conception de l'API, y compris ses points d'extrémité, ses modèles de données et les interactions entre eux. Une architecture d'API bien conçue peut améliorer les performances, la scalabilité et la facilité de maintenance de l'API. En utilisant des outils tels que Swagger et OpenAPI, les équipes peuvent créer une architecture d'API détaillée qui peut être facilement partagée avec les autres membres de l'équipe et les parties prenantes. Cela peut aider à garantir que tout le monde est sur la même longueur d'onde et peut contribuer à la conception et au développement de l'API.

## L'approche API First et les outils OpenAPI

Dans une approche API-First, l'objectif est de fournir des API REST en commençant par la conception de leurs contrats d'interface Swagger, suivie de leur développement et de leur intégration dans le code source de l'API. Cela permet d'anticiper les besoins de nos utilisateurs d'API et de leur fournir des API métier génériques sous forme de produits réutilisables pour plusieurs utilisateurs. Cette approche diffère de l'approche Code-First, où les contrats d'interface Swagger sont générés après que le code de l'API a été développé et intégré.

1. Prenez un fichier de contrat d'interface Swagger au format YAML en entrée.
2. Générez automatiquement le code source de l'API à partir de ce fichier YAML.
3. Mettez en œuvre les différents contrôleurs d'API.

4. Générez le nouveau contrat d'interface Swagger en sortie en fonction du code source de l'API.
5. Les deux contrats, entrée et sortie, doivent être identiques.



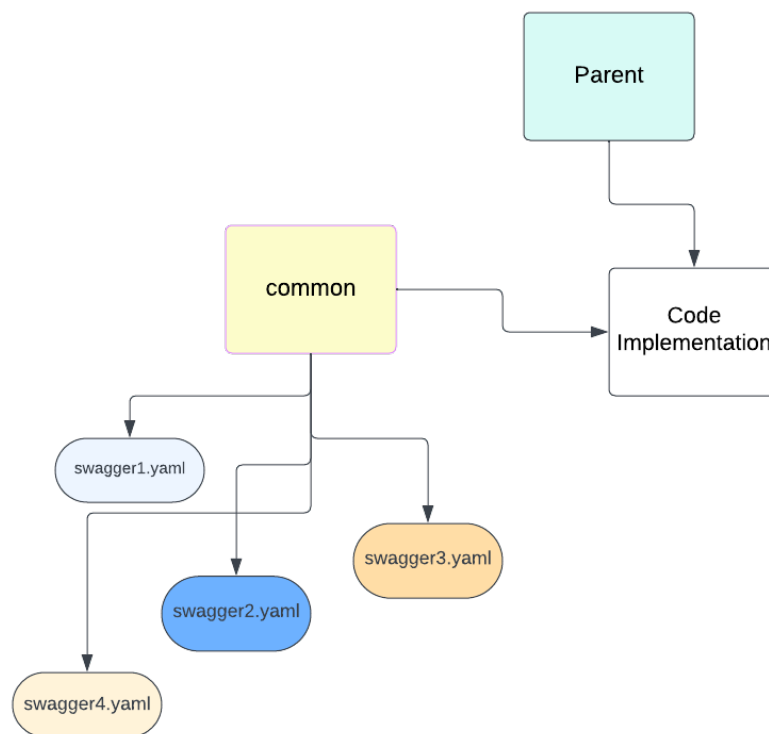
## Les outils OpenAPI

OpenAPI Tools est une collection d'outils et de bibliothèques open-source qui facilitent le travail avec les spécifications OpenAPI (anciennement connues sous le nom de Swagger). OpenAPI est un format de spécification qui vous permet de décrire des API REST de manière normalisée et lisible par les machines. L'écosystème des outils OpenAPI offre une série d'utilitaires pour aider dans diverses tâches liées au développement et à la documentation d'API.

Dans cette documentation, nous nous concentrerons sur les Outils de Génération de Code, qui fournissent des capacités de génération de code vous permettant de créer des brouillons côté serveur ou des kits de développement client en fonction de la spécification OpenAPI. Cela peut accélérer le processus de développement en créant automatiquement un code de base conforme au contrat de l'API.

## Générez la structure de code initiale en utilisant la fonction de génération de code des outils OpenAPI.

Dans notre situation, pour créer la structure de base qui englobe les interfaces et les configurations DTO, ainsi que pour utiliser les outils OpenAPI, nous avons besoin du contrat de l'API sous la forme d'un fichier Swagger fourni par le concepteur de l'API. Ensuite, le générateur produit le code en se basant sur ce contrat. Cependant, pour adapter notre générateur à nos besoins spécifiques, nous devons mettre en place un projet parent. Ce projet aura pour responsabilité de fournir toutes les propriétés personnalisées requises à notre générateur. Pour clarifier davantage, examinons attentivement ce schéma.



Comme vous pouvez le voir dans ce schéma, dans notre approche, nous avons choisi de centraliser la configuration sur le projet parent et de fournir cette configuration globale aux projets enfants. À ce stade, nous continuons de fournir les fichiers Swagger à partir

du dossier commun, mais c'est une approche à améliorer. Dans la prochaine section, nous aborderons d'autres approches pour fournir les fichiers Swagger de manière plus efficace.

Le schéma présente également le flux de travail et les composants impliqués dans l'utilisation des outils OpenAPI et de la génération de code pour faciliter le développement d'API basé sur le contrat Swagger.

**Concepteur d'API:** Le concepteur d'API joue un rôle crucial en créant le contrat Swagger (Swagger.yaml). Ce contrat sert de spécification pour l'API, définissant ses points d'extrémité, ses formats de requêtes/réponses, ses paramètres et autres détails.

**Ce projet commun** est responsable de la fourniture de tous les fichiers et éléments communs nécessaires à chaque projet, y compris la gestion des exceptions, l'intégration de Keycloak et plus encore. Le Swagger.yaml agit comme un accord standardisé entre le concepteur d'API et l'équipe de développement, garantissant que toutes les parties sont alignées sur les spécifications et la fonctionnalité de l'API. Il sert de base pour le processus de développement, guidant la mise en œuvre de l'API et assurant la cohérence entre différents projets.

**Le projet parent** dans une configuration Spring Boot est un projet POM Maven responsable de l'inclusion des dépendances communes, de la gestion des configurations et du processus de construction et d'emballage. Il simplifie la gestion des dépendances, favorise la réutilisation du code, garantit des constructions cohérentes et établit une structure de projet normalisée. En centralisant ces aspects, le projet parent améliore l'efficacité du développement, la qualité du code et la collaboration au sein de l'écosystème du projet.

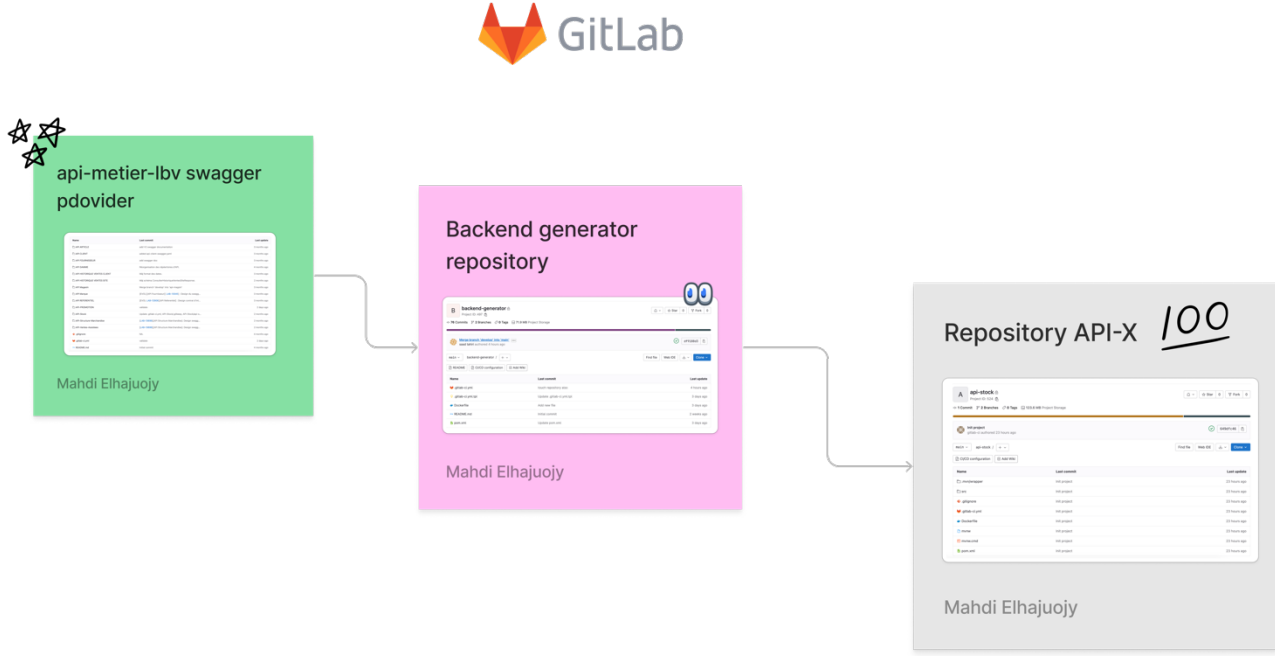
**Le projet d'implémentation** du code se concentre sur la mise en œuvre de la logique métier, la gestion des opérations sur les données, la définition des points d'extrémité de l'API et toute autre fonctionnalité spécifique à l'application. Il s'appuie sur le projet parent pour les configurations communes, les dépendances et les paramètres de construction définis à un niveau supérieur. De plus, il tire parti du projet commun pour la fonctionnalité partagée, telle que la gestion des exceptions, l'intégration de Keycloak et d'autres éléments nécessaires à plusieurs projets.

Le projet d'implémentation du code permet aux développeurs de se concentrer sur les exigences spécifiques de leur application tout en tirant parti des composants normalisés

et réutilisables fournis par les projets parent et commun. En séparant les préoccupations et en favorisant la modularité du code, il facilite la maintenance, améliore la réutilisabilité du code et favorise une approche de développement plus organisée et évolutive.

## Résumé

Pour résumer cette section, nous avons maintenant une compréhension générale de la manière dont fonctionnent les outils OpenAPI et comment nous pouvons générer un code de base basé sur notre contrat d'API. Ensuite, nous avons vu comment nous avons centralisé la configuration sur le projet parent, puis nous avons discuté du projet commun en tant que conteneur de notre fichier Swagger. Cependant, nous avons noté que l'idée de centraliser nos fichiers sur le projet commun n'est pas la meilleure approche, et nous chercherons une solution dans la prochaine section.





## API-Metier-lbv fournisseur Swagger

Maintenant, si vous examinez notre architecture, tout commence avec le concepteur d'API. Lorsqu'une nouvelle API est conçue ou mise à jour dans le référentiel:

<https://git.label-factory.ma/digital-referentiel-apis/api-metier-lbv>

Ce référentiel contient toutes les références des API de l'entreprise.

Dans le référentiel api-metier-lbv, qui est responsable de fournir les fichiers Swagger, un fichier gitlab-ci.yml sera déclenché à chaque commit :

```
stages:
  - validate
  - generate

validate-swagger:
  stage: validate
  image: node:lts-slim
  script:
    - npm install -g @apidevtools/swagger-cli
    - swagger-cli validate ./API-PROMOTION/api-promotion-swagger.yaml
    - cp ./API-PROMOTION/api-promotion-swagger.yaml open-api.yaml
  artifacts:
    paths:
      - open-api.yaml

bridge:
  variables:
    API_NAME: $API_NAME
  stage: generate
  trigger:
    project: aws-ci/digital/sandbox/backend-generator
    branch: develop
```

Le pipeline comprend deux étapes principales :

1. L'étape de validation du Swagger (validate-swagger stage)
2. L'étape du bridge (bridge stage)

Lorsque l'API est validée avec succès par l'étape de validation du Swagger, nous passons à l'étape suivante qui est responsable de déclencher un pipeline en downstream vers le pipeline backend-generator.

Pour clarifier davantage, dans le référentiel des références API, lorsque des validations sont effectuées, le pipeline commence à s'exécuter. Une fois que l'étape de validation est terminée, l'étape de pont déclenche également le démarrage du pipeline backend-generator. Cette action est appelée un pipeline en downstream.

Un pipeline en downstream est tout pipeline GitLab CI/CD déclenché par un autre pipeline.

## Note :

L'état actuel du projet fournisseur Swagger sert de déclencheur pour le pipeline backend-generator. Cependant, avec une approche plus efficace, le fournisseur Swagger pourrait également livrer directement le fichier Swagger au générateur backend. Cela nous permettrait de générer le référentiel en fonction de ce fichier et d'établir l'entrée "input.swagger" sur le générateur d'outils OpenAPI. Comme mentionné précédemment, nous avons fourni nos fichiers Swagger à partir du projet commun. Cependant, cette approche n'est pas optimale, car le projet commun devrait idéalement se limiter à partager uniquement des configurations et des fichiers communs, tels que la gestion des erreurs.

À l'avenir, bien que nous continuions avec l'approche commune pour le moment, notre plan est de passer à la fourniture du fichier Swagger depuis le projet de référence API à l'avenir. Il est important que les développeurs sachent que nous utilisons toujours le projet commun pour centraliser nos fichiers Swagger. En fin de compte, l'élément clé à retenir est que nous utilisons actuellement le fichier Swagger provenant du projet commun et l'utilisons pour déclencher avec précision le générateur backend dans ce projet.


# Backend generator CI/CD




Dans cette section, nous plongerons en profondeur dans les rouages du générateur backend, qui est responsable de la création du projet de démarrage de tout projet qui sera implémenté à l'avenir en fonction du fichier Swagger fourni.

Pour suivre la documentation, rendez-vous sur le générateur backend sur GitLab :


<https://git.label-factory.ma/aws-ci/digital/sandbox/backend-generator>



aws-ci > ... > sandbox > backend-generator

**backend-generator**  
Project ID: 497


  Star 0  Fork 0

66 Commits 2 Branches 0 Tags 71.2 MB Project Storage






 **Update .gitlab-ci.yml file**  
saad tahiri authored 1 day ago

 b8d77737 

main backend-generator / +

Find file Web IDE  Clone

README CI/CD configuration Add Wiki

Name	Last commit	Last update
 .gitlab-ci.yml	Update .gitlab-ci.yml file	1 day ago
 .gitlab-ci.yml.tpl	Update .gitlab-ci.yml.tpl	1 day ago
 Dockerfile	Add new file	2 days ago
 README.md	Initial commit	2 weeks ago
 pom.xml	Update pom.xml	2 days ago

## Fondement du Générateur - .gitlab-ci.yml :

maintenant en profondeur dans le générateur backend qui est responsable de la création de la structure que nous avons expliquée précédemment. Voyons comment le générateur backend fonctionne en détails, pour expliquer le fonctionnement du backend, voici la liste suivante :

En suivant l'approche du générateur backend, tout se passe dans le pipeline CI/CD. Examinons d'abord le pipeline, puis nous expliquerons chaque étape en détail.

Après avoir vérifié le fichier .gitlab-ci.yml

```
1 include:
2   - project: software-factory/automation/pipeline-templates/ci
3     file:
4       - /java/install.yml
5
6 variables:
7   API_NAME: 'api-stock'
8   PROJECT_PATH: /builds/aws-ci/digital/sandbox/backend-generator
9   REPOSITORY_URL: https://gitlab-ci-token:${GITLAB_CI_TOKEN}@CI_SERVER_HOST/aws-ci/digital/api/${API_NAME}.git
10
11 stages:
12   - generator
13   # - install
14   - update-repository
15
16 generate:
17   stage: generator
18   image: nexus.label-factory.ma:8012/maven:3.8.4-openjdk-17
19   script:
20     - microdnf install -y dnf-utils unzip curl zip bash wget
21     - wget -qO /usr/local/bin/yq https://github.com/mikefarah/yq/releases/latest/download/yq_linux_amd64
22     - chmod a+x /usr/local/bin/yq
23     - yq --version
24     - curl -O https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-cli/3.1.1/spring-boot-cli-3.1.1-bin.zip
25     - unzip spring-boot-cli-3.1.1-bin.zip
26     - cd spring-3.1.1/bin/
27     - ./spring init --build=maven --java-version=17 --packaging=jar --package-name=ma.lbv.api --group-id=ma.lbv.api --artifact-id=${API_NAME} --i
28     - unzip ${API_NAME}.zip -d ${API_NAME}
29     - cp -r ./${API_NAME} $PROJECT_PATH
30     - cd $PROJECT_PATH
31     - sed -ie "s/${API_NAME}/${API_NAME}/" pom.xml
32     - cat pom.xml
33     - cp pom.xml ${API_NAME}
34     - cp .gitlab-ci.yml.tpl ${API_NAME}/.gitlab-ci.yml
35     - sed -ie "s/${API_NAME}/${API_NAME}/" ${API_NAME}/.gitlab-ci.yml
36     - cp Dockerfile ${API_NAME}/Dockerfile
37     - rm -rf ${API_NAME}/.gitlab-ci.yml
38     - cd ${API_NAME}
39     - cd src/main/java/ma/lbv/api
40     - mkdir controller repository config mapper service entity
```

Dans ce pipeline, il y a deux étapes principales que nous expliquerons ultérieurement:

- Étape du générateur (generator stage)
- Étape de mise à jour du référentiel (update-repository stage)

L'étape du générateur concerne la création du projet de démarrage en utilisant l'image **openJdk**, ce qui nous permet de configurer l'environnement. Ensuite, c'est le moment de générer le projet.

Dans ce pipeline, nous avons de nombreux scripts, mais nous expliquerons les plus importants.

Le premier script important est le suivant:

```
. /spring init --build ...
```

Ce script est utilisé pour récupérer le Spring CLI, ce qui nous permet de créer un projet de démarrage en utilisant uniquement les options en ligne de commande fournies par Spring. Pour plus de détails, veuillez consulter la documentation officielle.

Nous avons fourni nos options qui seront prises en compte lors de la création du projet, comme le "group-id", "artifact-id", et d'autres.

Ensuite, les étapes suivantes sont également très importantes, car elles se concentrent uniquement sur la préparation du projet principal. La seule chose sur laquelle vous devez vous concentrer est l'étape où nous copions le contenu du fichier pom.xml à la racine de notre référentiel principal du générateur backend vers notre projet de démarrage. Dans ce cas, le pom.xml à la racine joue un rôle de modèle partagé parmi les API générées à l'avenir.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <!--      labelvie custom parent pom project  -->
  <parent>
    <groupId>ma.lbv.api</groupId>
    <artifactId>lbv-api-parent-poc</artifactId>
    <version>0.0.5-SNAPSHOT</version>
    <relativePath/>
  </parent>

  <groupId>ma.lbv.api</groupId>
  <artifactId>${API_NAME}</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>${API_NAME}</name>
  <description>${API_NAME}</description>
  <properties>
    <java.version>17</java.version>
    <api.name>${API_NAME}-swagger.yaml</api.name>
  </properties>
  <dependencies>


    <dependency>
      <groupId>ma.lbv.api</groupId>
      <artifactId>lbv-api-common-poc</artifactId>
      <version>0.0.8-SNAPSHOT</version>
    </dependency>


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

Ceci est le fichier pom.xml modèle qui se trouve à la racine du référentiel backend-gen. Vous pouvez en conclure que les variables ici seront fournies dans les variables du pipeline, et vous avez raison dans ce cas.

# API\_X Repository









aws-ci > ... > api > api-promotion > Repository

 **Update .gitlab-ci.yml file**  
saad tahiri authored 2 days ago

✓ d2985935 

main ▾ api-promotion / + ▾

HistoryFind fileWeb IDE⬇ ▾Clone ▾

Name	Last commit	Last update
 .mvn/wrapper	Init project	3 days ago
 src	Init project	3 days ago
 .gitignore	Init project	3 days ago
 .gitlab-ci.yml	Update .gitlab-ci.yml file	2 days ago
 Dockerfile	Init project	3 days ago
 mvnw	Init project	3 days ago
 mvnw.cmd	Init project	3 days ago
 pom.xml	Init project	3 days ago

Voici un exemple d'API générée et poussée vers un nouveau référentiel sur l'instance GitLab. Consultez l'exemple via ce lien :

<https://git.label-factory.ma/aws-ci/digital/api/api-promotion>

Comme nous l'avons expliqué, le générateur backend est responsable de tout générer. Les développeurs auront par défaut une instance de référentiel où, lorsqu'une nouvelle API est fournie, ils se concentreront uniquement sur la logique métier.

Tout est configuré selon la conception, les développeurs ne seront pas submergés par la structure, la création de fichiers Docker, les pipelines, etc.

Le projet de base généré par le backend sera fourni aux développeurs, comprenant:

- .gitlab-ci.yml
- Dockerfile
- pom.xml
- Project structure

## Résumé :

En résumé, le générateur backend joue un rôle crucial dans la simplification du processus de développement d'API. En fournissant aux développeurs un projet de base préconfiguré, il leur permet de se concentrer principalement sur la logique métier de l'API, tout en évitant les tracas liés à la mise en place de la structure, la création de fichiers Docker, les pipelines et autres éléments techniques. Grâce à l'utilisation du modèle pom.xml et à la gestion des variables de pipeline, le générateur backend crée efficacement des projets personnalisés pour chaque API générée, offrant ainsi une approche uniforme et efficace pour le développement. Cela libère les développeurs des tâches fastidieuses et leur permet de se concentrer sur la création de solutions innovantes, tout en garantissant cohérence et qualité dans l'ensemble du processus de développement d'API.

# Commencer et Installation

Pour commencer à utiliser ou à mettre en œuvre le projet généré par le générateur backend et poussé vers le groupe [aws-ci/digital/api/api-Name](https://git.label-factory.ma/aws-ci/digital/api/api-name), vous aurez besoin d'un compte GitLab et d'un accès au groupe en tant que membre de l'équipe DevOps. Si vous n'avez pas cet accès, vous devrez disposer de Git sur votre compte personnel. Une fois cela en place, procédez à la clonage du projet sur votre ordinateur personnel.

```
- git clone https://git.label-factory.ma/aws-ci/digital/api/api-name
```

Après avoir cloné et configuré Nexus sur votre ordinateur local pour la première et dernière fois, ouvrez le code source dans IntelliJ IDEA ou votre IDE préféré. Vous devrez également avoir OpenJDK 17 ou une version ultérieure.

Ne vous précipitez pas, nous coderons très bientôt. Donc, après avoir cloné le projet et ouvert le projet sur votre ordinateur personnel, nous allons maintenant expliquer comment vous pouvez implémenter le code. Avant cela, exécutez la commande suivante:

```
- mvn clean install -U
```

Cette commande est responsable de nettoyer le projet à partir du dossier "target", puis d'installer un nouveau dossier "target" avec les interfaces générées et les schémas (DTO) ainsi que la configuration de base, comme nous l'avons expliqué précédemment.

Si vous exécutez la commande et que les choses ne se déroulent pas comme prévu, vous rencontrez probablement un problème. Je suppose que le problème est causé par la configuration de Nexus sur votre ordinateur portable. Vous avez donc besoin d'une chose de plus pour que tout fonctionne comme prévu.

## Nexus Configuration:

Pour configurer Nexus sur votre ordinateur portable, vous devrez obtenir les permissions Nexus. Contactez l'équipe DevOps pour être ajouté à Nexus. Une fois ajouté, suivez ces étapes pour configurer Nexus sur votre environnement local.



```
- cd .m2
```

Après avoir accédé au dossier m2, vous devrez ajouter le fichier settings.xml s'il n'existe pas déjà dans le répertoire.

Ouvrez le fichier settings.xml dans votre IDE préféré, puis copiez et collez le bloc de code suivant:

```
<settings

  xmlns="http://maven.apache.org/SETTINGS/1.0.0"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
https://maven.apache.org/xsd/settings-1.0.0.xsd">

  <servers>

    <server>

      <!-- this id should match the id of the repo server in pom.xml --
>

      <id>nexus</id>

      <username>*****</username>

      <password>*****</password>

    </server>

  </servers>

  <pluginGroups>

    <pluginGroup>org.sonarsource.scanner.maven</pluginGroup>

  </pluginGroups>

  <profiles>

    <profile>

      <id>nexus</id>
```

```
<!--Enable snapshots for the built in central repo to direct -->

<!--all requests to nexus via the mirror -->

<repositories>

    <repository>

        <id>nexus</id>

        <url>https://nexus.label-
factory.ma:8443/repository/maven-public/</url>

        <releases>

            <enabled>true</enabled>

        </releases>

        <snapshots>

            <enabled>true</enabled>

        </snapshots>

    </repository>

</repositories>

<pluginRepositories>

    <pluginRepository>

        <id>nexus</id>

        <url>https://nexus.label-
factory.ma:8443/repository/maven-public/</url>

        <releases>

            <enabled>true</enabled>

        </releases>

        <snapshots>

            <enabled>true</enabled>
```

```
        </snapshots>

        </pluginRepository>

    </pluginRepositories>

</profile>

<profile>

    <id>sonar</id>

    <properties>

        <sonar.host.url></sonar.host.url>

    </properties>

</profile>

</profiles>

<activeProfiles>

    <activeProfile>sonar</activeProfile>

    <activeProfile>nexus</activeProfile>

</activeProfiles>

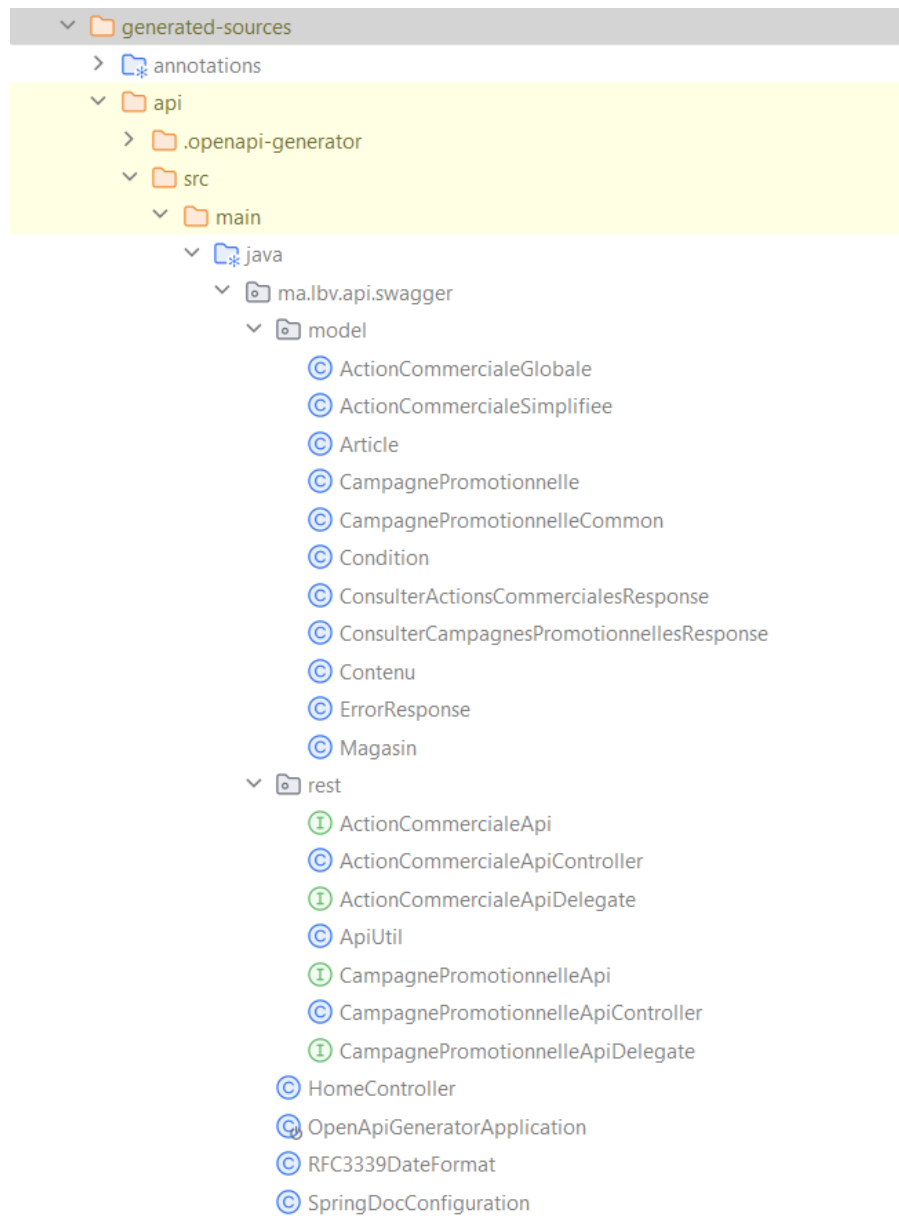
</settings>
```

Le fichier settings.xml configure une installation de Maven. Il est similaire à un fichier pom.xml, mais est défini de manière globale ou par utilisateur.

Consultez cet article pour comprendre chaque élément des paramètres que nous avons configurés.

<https://www.baeldung.com/maven-settings-xml>

## Générée Target



Voici une capture d'écran de la cible générée lorsque nous avons exécuté la commande `mvn clean install`.

Dans la configuration parent, nous spécifions tout, comme les noms des répertoires et où placer les modèles qui jouent le rôle de nos DTO, REST.

model: comprend les DTO ou schémas créés dans le swagger.yaml.

rest: nos interfaces de contrôleur et délégués ainsi qu'ApiUtil. Comme vous pouvez le deviner, des noms générés sont attribués aux interfaces et contrôleurs. Ces noms sont spécifiés dans la configuration pour les nommer en fonction des noms de tags dans le fichier Swagger. Le nom par défaut est le titre du fichier Swagger suivi de "Api". Chaque interface ou contrôleur, ainsi que chaque délégué, inclut tous les points d'extrémité de groupe tag à l'intérieur.

SpringDocConfiguration: dans cette classe, le générateur génère les informations d'API telles que le titre, les serveurs, les contacts, etc.

Enfin, tout est bien créé. Il ne reste plus qu'à expliquer comment le développeur peut utiliser ou implémenter les interfaces générées par le générateur d'outils OpenAPI.

## Résumé:

En somme, en suivant le processus de génération et de configuration que nous avons détaillé, vous avez maintenant une vision claire du générateur backend et de son rôle essentiel dans la création de projets de démarrage prêts à l'emploi. Grâce à l'utilisation d'outils OpenAPI, les interfaces, les contrôleurs, les DTOs et d'autres composants sont générés automatiquement à partir du fichier Swagger, éliminant ainsi la nécessité d'effectuer des tâches fastidieuses manuellement.

La configuration soigneusement planifiée, comme la gestion des variables de pipeline, l'utilisation du modèle pom.xml, et l'intégration avec Nexus, assure une cohérence et une qualité dans le processus de génération et de développement. Vous pouvez rapidement démarrer en clonant le projet, en configurant Nexus et en exécutant la commande `mvn clean install`. Cela aboutit à une cible générée contenant les éléments nécessaires pour l'implémentation des APIs, tels que les interfaces, les contrôleurs et la configuration.

En fin de compte, le générateur backend simplifie considérablement le processus de développement d'API en fournissant une structure prête à l'emploi, permettant aux développeurs de se concentrer davantage sur la logique métier et l'innovation.

# L'implémentation dans Spring Boot

Dans cette section, nous allons examiner de près la mise en œuvre en suivant cette approche que nous avons déjà exploitée auparavant. Je suis enthousiaste de partager davantage de façons dont vous pouvez mettre en œuvre cette approche de manière efficace.



À partir des interfaces et délégués générés, un développeur peut utiliser les classes générées comme point de départ pour la phase de développement. Par exemple, nous pouvons mettre en œuvre l'interface `CampagnePromotionnelleApiDelegate` en utilisant la classe `CampagnePromotionnelleController` ci-dessous :

```

@RestController

public class CampagnePromotionnelleController implements
CampagnePromotionnelleApiDelegate {

    private CampagnePromotionnelleService campagnePromotionnelleService;

    public CampagnePromotionnelleController(CampagnePromotionnelleService
campagnePromotionnelleService) {

        this.campagnePromotionnelleService = campagnePromotionnelleService;
    }

    @Override

    public ResponseEntity<ConsulterCampagnesPromotionnellesResponse>
consulterCampagnesPromotionnelles

        (LocalDate dateDebut, LocalDate dateFin,

        String dataSource, String codeCaisse, Integer page, Integer size) {

        return ResponseEntity.ok(

            campagnePromotionnelleService.consulterCampagnesPromotionnelles(

                dateDebut,dateFin,dataSource,codeCaisse,page,size

            )

        );

    }

}

```

Dans notre approche, nous avons choisi d'utiliser le délégué à implémenter. Qu'est-ce que le modèle délégué (delegate pattern) ?

## Modèle délégué dans le code généré par Swagger?

Tout d'abord, une clarification: vous n'êtes pas obligé d'utiliser le modèle de délégation. Au contraire, le comportement par défaut du générateur Spring est de ne pas utiliser le modèle de délégation, comme vous pouvez le vérifier facilement dans la documentation. Dans ce cas, il générera uniquement l'interface `CampagnePromotionnelleApi` et `CampagnePromotionnelleApiController`.

### Pourquoi utiliser la délégation?

Cela vous permet d'écrire une classe qui implémente `CampagnePromotionnelleApiDelegate`, qui peut être facilement injectée dans le code généré, sans avoir besoin de toucher manuellement aux sources générées. Cela préserve la mise en œuvre des éventuels changements futurs dans la génération de code.

Pensons à ce qui pourrait se passer sans la délégation.

Une approche naïve serait de générer les sources, puis d'écrire directement la mise en œuvre à l'intérieur de `CampagnePromotionnelleApiController`. Bien sûr, la prochaine fois qu'il sera nécessaire d'exécuter le générateur, cela serait problématique. Toute la mise en œuvre serait perdue...

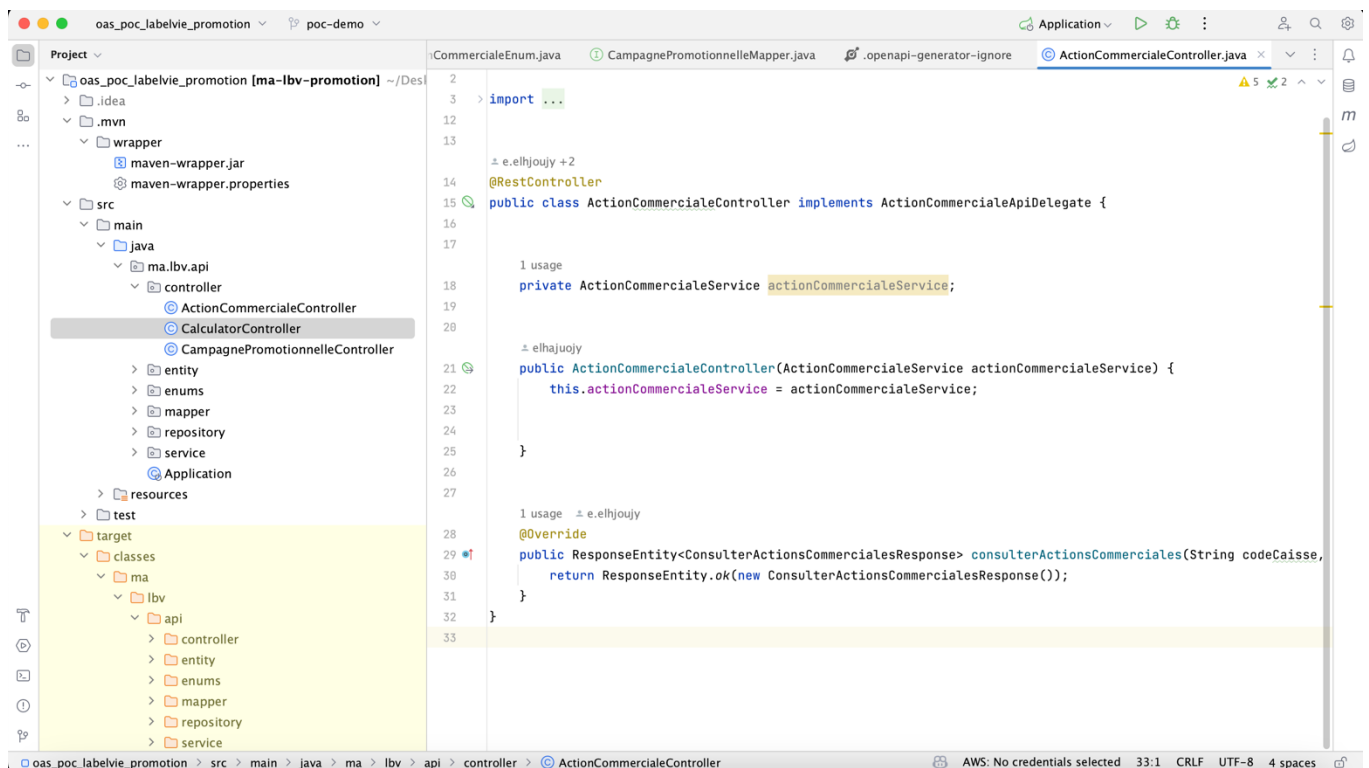
Un scénario légèrement meilleur, mais pas parfait, serait d'écrire une classe qui étend `CampagnePromotionnelleApiController`. Cela protégerait la mise en œuvre contre les réécritures pendant la génération, mais ne la protégerait pas contre les futures modifications du moteur de génération: au minimum, la classe de mise en œuvre devrait implémenter le constructeur de `CampagnePromotionnelleApiController`. Actuellement, le constructeur d'un contrôleur généré a la signature suivante: `CampagnePromotionnelleApiController (ObjectMapper objectMapper, HttpServletRequest request)`, mais les développeurs du générateur pourraient avoir besoin de le changer à l'avenir. Donc, la mise en œuvre devrait également changer.

Une troisième approche serait d'oublier complètement `CampagnePromotionnelleApiController` généré et d'écrire simplement une classe qui implémente l'interface `CampagnePromotionnelleApi`. Ce serait bien, mais à chaque génération du code, vous devriez supprimer manuellement



CampagnePromotionnelleApiController, sinon le routeur Spring généré se plaindrait. Toujours du travail manuel ...

Mais avec la délégation, le code de mise en œuvre est complètement sûr. Pas besoin de supprimer manuellement des éléments et pas besoin de s'adapter en cas de changements futurs. De plus, la classe qui implémente CampagnePromotionnelleApiDelegate peut être traitée comme un service indépendant, vous permettant d'injecter / d'auto-câbler tout ce dont vous avez besoin.



Bien que le générateur ne puisse pas produire directement notre logique métier, il fournit une base solide. Ce qui vient ensuite, c'est la mise en œuvre de notre logique métier. Comme le montre cette capture d'écran, nous commençons d'abord par mettre en œuvre le délégué, puis nous passons à la création de notre logique métier.

## Tests de l'implémentation du code sur l'application Spring avec Postman :

The screenshot shows the Postman interface with a GET request to the endpoint `{{baseUrl}}/v1/campagnes-promotionnelles?codeCaisse=GH77KQj&dateDebut=2020-10-10&dateFin=2020-12-12&page=1&size=50`. The request is successful, returning a 200 OK status with a response time of 312 ms and a size of 714 B.

The response body is displayed in JSON format, showing an array of promotional campaigns:

```
1 {
2   "campagnesPromotionnelles": [
3     {
4       "titre": "Titre 1",
5       "numero": 23,
6       "dateDebutPromo": "2022-10-12",
7       "dateFinPromo": "2022-12-12",
8       "magasinsConcernes": [],
9       "actionsCommerciales": []
10    },
11    {
12      "titre": "Titre 2",
13      "numero": 13,
14      "dateDebutPromo": "2023-10-12",
15      "dateFinPromo": "2023-12-12",
16      "magasinsConcernes": [],
17      "actionsCommerciales": []
18    }
19  ]
20 }
```

Comme vous pouvez le constater, le point de terminaison fonctionne très bien .

## Conclusion:

En résumé, l'approche que nous avons explorée dans ce document met en avant l'utilisation de la génération de code basée sur Swagger pour accélérer le processus de développement d'API. En priorisant la conception d'API en amont et en utilisant des outils tels que Swagger et OpenAPI, nous avons pu créer une base solide de composants générés automatiquement, tels que les interfaces, les contrôleurs et les schémas.

Le générateur backend nous permet de déléguer la création de la structure et des interfaces d'API à l'automatisation, nous libérant ainsi du fardeau de créer manuellement des tâches fastidieuses. L'approche déléguée garantit que la mise en œuvre est protégée des futurs changements dans la génération de code, tout en offrant la flexibilité nécessaire pour implémenter notre propre logique métier.

L'intégration de Nexus assure une gestion centralisée des dépendances, ce qui garantit la cohérence et la qualité de chaque projet généré. De plus, la configuration soigneusement planifiée des pipelines CI/CD facilite le flux de travail du développement jusqu'au déploiement.

L'implémentation de l'approche dans Spring Boot a été détaillée, mettant en évidence l'importance de la mise en œuvre des délégués pour maintenir la flexibilité et la protection de la logique métier. En utilisant les classes générées comme point de départ, nous pouvons rapidement construire des fonctionnalités spécifiques tout en évitant les tracas de la création manuelle de code répétitif.

En fin de compte, cette approche présente une solution efficace pour accélérer et améliorer le processus de développement d'API, tout en garantissant la qualité, la cohérence et la flexibilité nécessaires pour répondre aux besoins évolutifs de l'entreprise. En combinant automatisation, délégation et mise en œuvre ciblée, nous pouvons relever avec succès les défis du développement d'API et créer des produits plus performants et plus évolutifs.