TP Kotlin: Création d'un controller Spring à l'aide de KSP

A travers ce TP, nous allons reprendre une partie du TP sur notre chapitre "Spring". Si vous n'avez pas fait ou pas fini, il est recommandé de finir le TP sur Spring et ensuite de revenir sur ce TP si vous avez terminé. L'objectif sera de recréer le controller "VehicleController".
Rien ne vous empêche de regarder le code du controller ! Après tout, ce n'est pas sur ça qu'on se focalise ! On se focalise sur comment nous allons "automatiser" la création d'un Controller.

(Pourquoi réécrire une partie d'un TP ?) : Pour démontrer l'utilité de KSP dans la gestion de code répétitif, en effet, le code d'un controller est souvent répétitif et similaire à celui d'un autre controller(opérations CRUD classique le plus souvent).
C'est pourquoi nous allons faire ce controller avec KSP.

```
package fr.esgi.generated.controllers

// Automatically generated by KSP
import org.springframework.http.HttpStatus
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.*

import fr.esgi.Vehicle
@RestController
@RequestMapping("api/vehicle")
class VehicleController(private val vehicleRepository :
VehicleRepository) {

      @PostMapping
      @ResponseStatus(HttpStatus.CREATED)
      fun createVehicle(@RequestBody vehicle: Vehicle): Vehicle =
vehicleRepository.save(vehicle)

      @GetMapping
      fun getAllVehicles(): List<Vehicle> = vehicleRepository.findAll()

      @GetMapping("/{id}")
      fun getVehicleById(@PathVariable id: Long): Vehicle? =
      vehicleRepository.findById(id).orElse(null)

      @PutMapping("/{id}")
      fun updateVehicle(@PathVariable id: Long, @RequestBody vehicle:
Vehicle): Vehicle? {
      return if(vehicleRepository.existsById(id)){
            vehicleRepository.save(vehicle.copy(id = id))
      } else null
      }
```

```
    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    fun deleteVehicle(@PathVariable id: Long) =
vehicleRepository.deleteById(id)
}
```

I) Initialisation du TP

Reprenez la structure du code se trouvant sur le lien github suivant:
https://github.com/ari1008/fyc_2024_gen_kot
Rendez-vous sur la branche “KSP/TP”
Rajoutez les dépendances Spring:

```
plugins {
    kotlin("jvm") version "1.9.25"
    kotlin("plugin.spring") version "1.9.25"
    id("org.springframework.boot") version "3.3.5"
    id("io.spring.dependency-management") version "1.1.6"
}

dependencies {
    implementation("org.springframework.boot:spring-boot-starter-web")
    implementation("org.jetbrains.kotlin:kotlin-reflect")

testImplementation("org.springframework.boot:spring-boot-starter-test")
}
```

Vous devriez normalement avoir cela ET ce qui était dans la structure du code KSP de base.
N’oubliez pas de build le projet !

II) Création de/des annotation(s)

Vous allez maintenant devoir créer vos potentiels annotations dans le module “annotations”

Rappelez-vous, quel est le type de target que l'on cherche à cibler dans le cas de la création d'un controller Spring ?(ou plus généralement quelle structure de données on cherche à envoyer/traiter).

```
import jakarta.persistence.Entity
import jakarta.persistence.GeneratedValue
import jakarta.persistence.GenerationType
import jakarta.persistence.Id
Ajoutez votre annotation la dedans quelque part(il peut etre interessant
de rajouter le chemin de votre mapping dans les args de l'annotation)
@Entity
data class Vehicle(
      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      val id: Long = 0,
      val brand: String,
      val model: String,
      val vehicleYear: Int
) {
      constructor() : this(0, "", "", 0) {

      }
}
```

III) Création du processor

Créez dès à présent votre fichier Processor et ProcessorProvider, n'oubliez pas le fichier dans le dossier "ressources" ainsi que son contenu.

ControllerProcessorProvider

Créez la structure classique d'un Provider, aucun réel changement à ce niveau.

ControllerProcessor

process()

- Récupérez les symbols avec `resolver.getSymbolsWithAnnotation()`
- Créez votre fichier généré ou vous le souhaitez
- Commencez l'écriture du fichier avec les packages et les imports(très important !)
- Pour chaque symbole on les fait passer à travers le Visitor
- Enfin créer votre inner class Visitor pour gérer chaque type de structure de données que vous allez rencontrer !

inner class ControllerVisitor()

Conseil: Commencez par la création de vos fonctions de façon globale, c'est à dire classe puis propriété.

- Afin de gérer les cas généraux d'abords(GETALL(), CREATE()) avec visitClassDeclaration().
- Puis ensuite les cas avec paramètres(GETBYID(), DELETEBYID(),PUT()) avec visitPropertyDeclaration().

Enfin n'oubliez pas:

- Libre à vous de concevoir votre propre façon pas obligatoire de faire l'inner class ou d'utiliser les fonctions de visitNameOfObject…()
- Comment allez vous gérer le Mapping ??