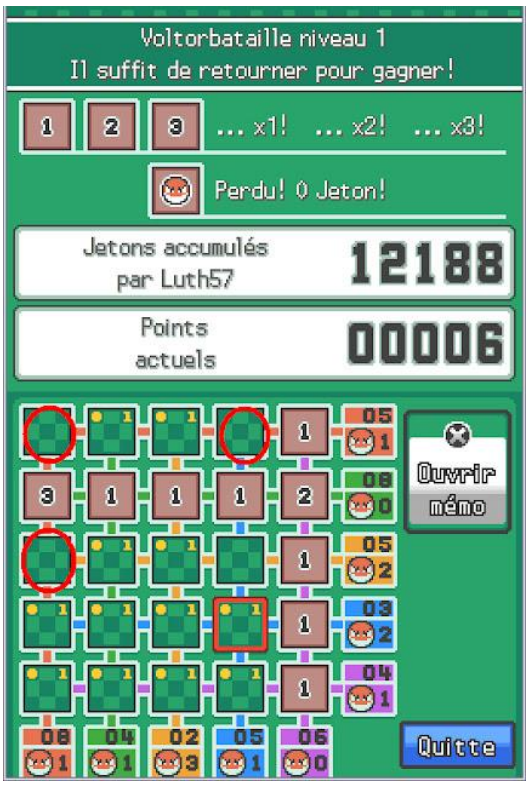
Voltorbataille

Exercice Java Spring

Table des matières

But de l’exercice



Le but de cet exercice va être de recréer ce jeu. Il s’agit du mini-jeu Voltorbataille présent dans certains jeu Pokémon de la 4ème génération (Heartgold et Soulsilver).

Le jeu génère une grille de 5x5 cases avec des points allant de 1 à 3 et des bombes. Au bout de chaque ligne et colonne est indiqué le nombre de points et le nombre de bombes de la ligne ou de la colonne.

A chaque fois que l’on clique sur une case elle est révélée, si ce sont des points qui sont révélé alors le score actuel est multiplié par le nombre de points de la case et si c’est une bombe alors on perd la partie.

Le but du jeu est de révéler toutes cases avec des 2 et des 3. Si la grille est résolue alors les points actuels sont ajoutés aux points déjà gagnés lors des grilles précédentes, puis on passe à la prochaine grille avec un niveau de difficulté augmenté de 1 et les points actuels remis à 0.

Au cours de cet exercice, si vous n’avez pas bien compris quelque chose ou que vous n’y arrivez pas, n’hésitez pas à vous inspirer du projet d’exemple.

Mise en place du projet

Vous trouverez dans ce dossier 2 projets :

- **api-grille-voltorbataille** : il s'agit de l'api que l'on va utiliser pour générer la grille.
- **exemple-back-voltorbataille** : il s'agit d'une solution d'exemple à cet exercice.

Commencez par vous assurer que vous pouvez lancer le projet **api-grille-voltorbataille** en lançant la commande « *mvn clean install* » pour build le projet puis en lançant le projet.

Création du projet avec Spring Initializr

Rendez vous sur https://start.spring.io/ pour créer votre projet. Renseignez les champs, nous voulons un projet **Maven**, en **Java 21**.

Ajoutez ensuite les dépendances suivantes :

- **Thymeleaf** (pour générer un front rapidement directement dans le projet java)
- **Spring Data JPA** (pour une éventuelle base de données)
- **MySql Driver**
- **Spring Web**
- **Docker Compose Support** (si vous souhaitez dockeriser le projet)
- **Testcontainers** (pour effectuer des tests avec docker)
- **Spring Security**
- **Spring Boot DevTools**
- **Lombock** (pour ajouter des annotations bien pratiques)

Générez ensuite le projet, téléchargez et décompressez le fichier obtenu et vous êtes prêt à vous lancer !

Petit détour par Thymeleaf

Thymeleaf permet de réaliser un front pour tester votre projet. Pour cela il suffit de créer une classe annotée avec **@Controller**. Dans votre navigateur il suffira de vous rendre aux urls renseignées dans cette classe pour afficher une page correspondant au template que retourne cette url. Les templates Thymeleaf sont stockés dans les ressources du projet, dans *./resources/templates*.

J'ai un peu la flemme d'expliquer comment fonctionne Thymeleaf, je ne suis pas super au point dessus et ce n'est pas vraiment le but de l'exercice. Cela dit ce n'est pas très compliqué, si vous souhaitez réaliser votre propre front je vous laisse vous renseigner sur https://www.thymeleaf.org/ .

Sinon vous pouvez juste récupérer les ressources du projet d'exemple et le package **exemple_back_voltorbataille.ui** du projet d'exemple.

Vous connaissez Spring ?

Si vous ne connaissez pas du tout Spring renseignez-vous un peu avant de vous lancer dans cet exercice.

Ce que vous devez surtout savoir c'est que Spring permet de gérer plus facilement des dépendances entre classes grâce à des annotations qui font de vos classes des **@Bean**. En annotant correctement vos classes vous pourrez ainsi créer des singletons pour vos services et les importer dans d'autres classes sans avoir à les initialiser à la main.

Créer une nouvelle partie

La classe partie

Nous souhaitons maintenant créer une nouvelle partie. Pour cela je vous laisse gérer l'organisation de votre projet, essayez au moins de le ranger mieux que votre chambre.

Dans tous les cas créez une classe **Partie** pour stocker les informations relatives à une partie, par exemple le nombre de points en cours, le nombre de points gagnés durant la partie, quelles cases sont révélées par le joueur, le niveau de difficulté, si la partie est gagnée ou perdue et surtout la grille (ce qui implique que vous devrez créer une classe Grille).

Une fois cela fait nous allons utiliser deux annotations de classe Lombok afin de nous faciliter la tâche.

La première est **@Data**, elle permet de générer automatiquement les getters et setters de la classe à la compilation du projet sans avoir à les coder et à pourrir son code avec des fonctions aussi simples.

La seconde est **@NoArgsConstructor**, elle permet de générer automatiquement un constructeur de la classe et initialisant tous ses attributs à null. Il existe deux autres annotations semblables **@AllArgsConstructor** qui génère un constructeur prenant en paramètre tous les attributs de la classe, et **@RequiredArgsConstructor** que nous utiliserons plus tard.

Voilà la classe Partie est terminée et elle est normalement bien lisible sans fonctions pour nous gâcher la vue.

Et la classe Grille ?

Effectivement il y a une grille dans la classe Partie, il va donc falloir lui faire une classe aussi. Mais il se trouve qu'elle existe déjà ! Dans l'api-grille-voltorbataille puisque c'est elle qui va nous la fournir.

Vous allez donc pouvoir récupérer les classes **Grille** et **Indice** dans le package **api_grille_voltorbataille.grille.model** et les ajouter à votre projet. On remarquera d'ailleurs que l'annotation @AllArgsConstructor aurait pue être utilisée dans la classe Indice, mais que je n'y ai pas pensé en la faisant.

Récupérer la grille

Pour récupérer cette grille il va falloir faire un appel à l'api_grille_voltorbataille. Pour cela nous allons créer un service qui s'en occupera. Créez une classe, vous pouvez l'appeler GrilleApiService si vous voulez me copier, et rangez la bien.

Puisque c'est un service nous allons l'annoter avec **@Service**, cette annotation permet faire de cette classe un singleton qui pourra être importé simplement dans d'autres classes. C'est dans les services que va se réaliser la logique et les traitements de votre application. Dans ce service vous allez donc pouvoir stocker toutes les fonctions qui vont gérer la connexion à l'api_grille_voltorbataille (il n'y en a qu'une).

L'url utilisée pour récupérer une grille est de la forme :

http://localhost:8081/grille?hauteur=5&largeur=5&difficulte=1

Où la hauteur, la largeur et la difficulté peuvent être changées. Vous pouvez tester cette url dans votre navigateur en lançant le projet api_grille_voltorbataille.

Mais peut-être souhaitez vous lancer cette api sur un serveur ? Ou sur un autre port ? Dans ce cas il va falloir stocker l'host et le port dans une variable du projet pour pouvoir la changer facilement en fonction des besoins.

Pour cela il y a deux fichiers dans les ressources du projet **application.properties** et **application-local.properties**.

Dans application-local.properties vous allez les variables API_GRILLE_HOST et API_GRILLE_PORT sous ce format :

```
API_GRILLE_HOST=localhost
API_GRILLE_PORT=8081
```

Ce sont ces valeurs qui vont être utilisées par votre application.

Puis dans application.properties vous allez stocker l'url complète vers l'api_grille_voltorbataille comme ceci :

```
grilleApi.url=http://${API_GRILLE_HOST:localhost}:${API_GRILLE_PORT:8081}
```

On voit que les variables définies plus tôt sont utilisées ici et que si elles ne sont pas renseignées dans application-local.properties elles utilisent les valeurs par défaut *localhost* et *8081*.

Bien il faut maintenant pouvoir récupérer cette url dans notre classe de service. Pour cela nous allons utiliser l'annotation **@Value** comme au-dessus de l'attribut de

classe dans lequel on souhaite stocker cette valeur. Par exemple dans la classe de Service :

```
@Value("${grilleApi.url}")
private String grilleApiUrl;
```

Ici on stocke la valeur de *grilleApi.url* définie dans *application.properties* dans l'attribut *grilleApiUrl* de la classe de service.

Il est maintenant temps de faire l'appel à l'api. Pour cela nous allons utiliser la classe **RestTemplate** qui permet de convertir le résultat d'un appel Rest directement en une classe. Ici nous allons convertir le résultat de l'appel pour récupérer une grille en **Grille** :

```
public Grille getGrille(int hauteur, int largeur, int difficulte) {
  RestTemplate restTemplate = new RestTemplate();
  return restTemplate.getForObject(grilleApiUrl + "/grille" + "?hauteur=" + hauteur +
        "&largeur=" + largeur + "&difficulte=" + difficulte, Grille.class);
}
```

Et voilà la grille est récupérée très facilement ! Cette fonction va ensuite devoir être appelée depuis un autre service, qui lui va gérer la logique des parties. Je vous laisse le créer ce **PartieService** avec l'annotation @Service.

Le PartieService

Ce service va gérer la logique des parties, notamment la création et le stockage d'une nouvelle partie. Nous allons en profiter pour voir quelques nouvelles annotations.

@SessionScope est une annotation que nous allons mettre sur la classe de service. Elle permet de définir la durée de vie d'un composant (d'un *bean*) à une session. Cela veut dire qu'un nouveau service sera créé pour chaque nouvelle session utilisateur du front, nous allons donc pouvoir stocker la partie en cours d'un utilisateur dans le *PartieService*, ainsi chaque utilisateur aura son propre service et donc sa propre partie jusqu'à ce que sa session expire. Je vous laisse ajouter cette annotation et gérer le stockage de la partie dans le service, c'est un exercice après tout, pas un tuto (indice : c'est un singleton).

Une fois que c'est fait nous allons pouvoir créer une fonction qui crée une nouvelle partie, cette fonction devra mettre tous les points à 0 et récupérer une nouvelle grille via le service de gestion de l'api_grille_voltorbataille. Vous allez donc devoir importer ce service dans *PartieService* et c'est là que Spring est fort ! Vous pouvez faire cela en utilisant l'annotation **@Autowired** comme ceci :

```
@Autowired
private final GrilleApiService grilleApiService;
```

L'annotation va ainsi prendre en charge l'import du service, l'appel au constructeur, etc.

Encore mieux, au lieu d'utiliser *@Autowired* sur tous les attributs que vous souhaitez importer, ce qui finirait par être moche dans un très gros service, vous pouvez définir tous ces attributs en *final* et utiliser l'annotation Lombok **@RequiredArgsConstructor** sur la classe de service qui va automatiquement générer un constructeur pour tous les attributs *final* de la classe et les initialiser !

A partir de là vous pouvez créer une fonction *nouvellePartie()* qui se charger d'effectuer toutes les opérations relatives à la création d'une nouvelle partie, notamment l'appel à d'autres services.

Les mappers

Votre partie peut maintenant être crée et il faudrait l'envoyer au front pour pouvoir jouer, mais vous ne voulez PAS envoyer toutes les infos de la partie au front sinon l'utilisateur pourrait tricher en lisant le résultat de la requête et en voyant à quoi ressemble la grille complète. Et vous savez quoi faire : un DTO, une classe qui ne contient que les infos utiles à l'utilisateur.

Créez une classe **PartieDto** avec **@Data** qui ne contient que les infos utiles au front dont la grille. Appelez les attributs de la même manière que dans la classe *Partie*.

Une fois cela fait il faut créer un mapper entre ces 2 classes pour cela nous devront utiliser Mapstruct (vous devrez peut être l'ajouter à votre projet : https://mapstruct.org/documentation/installation/)

Créez une interface **PartieMapper** et annotez-la avec **@Mapper(componentModel="spring")**. Grâce à cette annotation vous pourrez définir des fonctions comme :

```
public Partie partieDtoToPartie(PartieDto partieDto);
```

Sans avoir besoin de coder l'implémentation de la fonction. Mapstruct va automatiquement prendre les attributs ayant le même nom dans chaque classe et leur attribuer la même valeur.

SAUF QUE nous on veut que la grille du Dto ne contienne que les valeurs des cases révélées et -1 si la case n'est pas révélée. Il va donc falloir définir une fonction avec **default** :

```
default PartieDto partieToPartieDto(Partie partie)
```

Et implémenter manuellement le code qui converti une *Partie* en *PartieDto*. Il va donc falloir créer une nouvelle *PartieDto* et lui affecter ses attributs, notamment la grille modifiée.

Bonne chance allez-y.

Les Controller

BIEN, maintenant que vous avez brillamment réussi votre mapper il serait temps de voir comment récupérer les demandes de l'utilisateur. Pour cela on va passer par des Controller.

Si vous avez copié-collé le front de l'exemple vous avez dû voir le controller utilisé pour Thymeleaf. Vous savez donc déjà qu'il faut annoter sa classe de controller avec **@Controller** ou **@RestController** pour pouvoir réceptionner des requêtes. Mais dans un controller Thymeleaf il n'y a que des GET et ce n'est pas le cas pour la plupart. Nous, par exemple, on voudrait un GET pour récupérer les infos de la partie en cours (la PartieDto) et un POST pour créer une nouvelle partie, et même encore un autre pour cliquer sur une case.

Voyons d'abord le plus simple, le GET. Pour qu'une fonction puisse recevoir des requêtes Rest GET il faut l'annoter avec **@GetMapping**, par exemple :

```
@GetMapping("/partie")
public PartieDto getPartie()
```

Grâce à cette annotation cette fonction se déclenchera dès que l'utilisateur fera une requête GET à l'url http://localhost:8080/partie . A vous d'implémenter cette fonction.

Passons ensuite au POST. Tout comme GET a son annotation POST a la sienne : **@PostMapping**. Nous pouvons ainsi définir les fonctions suivantes :

```
@PostMapping("/partie")
public void nouvellePartie() {
  partieService.nouvellePartie();
}

@PostMapping("partie/click-on-case")
public void clickOnCase(@RequestParam int x, @RequestParam int y) {
  partieService.clickOnCase(x, y);
}
```

Vous remarquerez deux choses, qu'une annotation GET et POST peuvent avoir la même url mais que deux annotations du même type ne le peuvent pas. Ensuite on peut voir une nouvelle annotation **@RequestParam** qui permet de récupérer les infos de la requête, ici par exemple les *x* et *y* présents dans l'url de la requête. Si ces infos étaient stockées dans le body de la requête alors on aurait utilisé l'annotation **@RequestBody**.

FIN

ET BAM à partir de là vous savez tout ce qu'il y a à savoir pour finir le jeu ! pour l'instant…

Nous verrons par la suite comment ajouter des comptes utilisateur et de la sécurité avec Spring Security. Nous verrons aussi comment stocker ces utilisateurs dans une base de données et comment dockeriser le projet.

Ces étapes sont déjà commencées dans le projet d'exemple mais pas terminées vous pouvez y jeter un œil si vous êtes curieux.

Aller, je vous laisse finir de coder ce jeu. Je suis sympa je vous donne quelques étapes :

- Implémenter la logique pour révéler des cases
- Calculer les points
- Savoir si on a gagné ou perdu ou aucun des deux
- Passer au niveau suivant, regénérer une grille
- Regarder comment fonctionne Thymeleaf pour tester votre projet (pas compliqué promis)