Rapport technique du projet JEE

Equipe :
Alan DELY
Matteo GAILLARD
Ethan LEVACHER
Riane MBODJI

Année scolaire 2025-2026

Sommaire

1. Introduction du projet :

L'objectif est de refondre entièrement le site web d'un parc d'attractions afin d'améliorer à la fois la gestion interne et l'expérience des visiteurs.
Le travail se déroule en groupe et couvre l'ensemble de la démarche : analyse UML complète (acteurs, cas d'utilisation, diagrammes), conception d'une architecture cohérente puis développement d'une application web reposant sur un backend Java/Tomcat en REST, un frontend Vue JS et une base MySQL.
Le site doit proposer une partie publique présentant les attractions avec leur temps d'attente en temps réel grâce à une API externe, ainsi qu'un espace d'administration permettant de gérer attractions, spectacles, personnages et statistiques.

2. Analyse fonctionnelle et besoins techniques :

2.1) Le site doit permettre :

- Aux visiteurs de consulter les attractions, spectacles et personnages du parc,
- De connaître en temps réel les temps d'attente des attractions,
- Aux administrateurs du parc de gérer l'ensemble des contenus (attractions, Spectacles, rencontres, personnages)
- D'obtenir des statistiques d'activité.

2. 2) Les acteurs humains du système sont :

- Visiteur : consulte les attractions, spectacles et temps d'attente
- Administrateur : gère les attractions, spectacles, personnages, rencontres, statistiques.

2.3) Les acteurs systèmes sont :

- API externe de calcul des temps d'attente : fournit en temps réel le temps d'attente pour chaque attraction.
- Système de base de données : stocke tout le contenu administré.

2.4) Les contraintes fonctionnelles :

- Le temps d'attente qui doit être récupéré en temps réel via l'API externe.
- La gestion des spectacles
    - Un personnage ne peut pas être présent dans 2 spectacles qui se chevauchent.
    - Si deux spectacles se jouent dans des lieux différents : 30 minutes de battement entre la fin d'un spectacle et le début du suivant

- La gestion des personnages : chaque personnage est unique avec un seul costume et pas de double utilisation simultanée.
- Un personnage ne peut pas être en rencontre si :
    - il joue dans un spectacle au même moment,
    - ou si la rencontre viole la contrainte des 30 minutes entre deux lieux différents.
- Les statistiques
    - L'activité d'un personnage = somme des heures de spectacle + rencontres.
    - Classement → tri décroissant.

2.5) Use Case

Partie publique

1. Consulter la liste des attractions
2. Consulter le détail d'une attraction
3. Consulter les temps d'attente
4. Consulter les spectacles
5. Consulter les personnages d'un spectacle
6. Consulter les rencontres personnages

Partie admin

7. Gérer les attractions (CRUD)
8. Gérer les spectacles (CRUD)
9. Gérer les rencontres (CRUD)
10. Affecter des personnages aux spectacles
11. Vérifier les conflits horaires
12. Consulter les statistiques

(CRUD : Create, Read, Update, Delete)

3. Architecture générale du système

Pour l'architecture de notre back-end, nous avons utilisé la même organisation que celle étudiée en cours. Tout d'abord, nous créons des modèles pour représenter les données que nous utilisons (par exemple la classe Attraction). Ensuite, pour interagir avec ces données, nous utilisons des DAO (Data Access Object) : ce sont ces classes qui s'occupent de communiquer avec la base de données. Nous avons créé une interface DAO Interface contenant les méthodes de base que nous jugeons nécessaires d'implémenter dans chaque DAO :

- findAll()
- findById()
- create()
- delete()

Chaque DAO doit implémenter cette interface.

Nous avons ensuite les controllers, dans lesquels nous définissons les routes et appelons la bonne fonction du bon DAO pour effectuer une action. La vérification des paramètres se fait via le système de validation de Jakarta ainsi que Jersey Validation. Dans les DAO, nous déclarons également des record Java pour représenter les propriétés attendues pour les différents modèles.

Pour simplifier le développement, nous utilisons une classe DBRequest, qui fournit des méthodes facilitant les interactions avec la base de données, telles que :

```
public static <T> List<T> executeSelect(
        @NotNull String query,
        @Nullable Map<Integer, Object> params,
        @Nullable Function<ResultSet, T> callback
)
```

```
public static IdResponse executeCommand(
        @NotNull String query,
        @Nullable Map<Integer, Object> params
)
```

Concernant l'utilisation de l'API des temps d'attente, nous avons décidé de mettre en place une encapsulation interne afin de maîtriser totalement son exposition. Toutes les deux minutes, nous mettons à jour un cache contenant, pour chaque attraction, son temps d'attente. Lors de l'envoi des données d'une attraction, nous allons simplement récupérer le temps d'attente dans ce cache. Cela nous permet de limiter le nombre de requêtes vers l'API et d'éviter d'exposer directement celle-ci au client.

Enfin, pour gérer proprement les erreurs lors du déploiement ou de l'exécution de l'application, nous utilisons ExceptionMapper fourni par Jakarta, qui nous permet

d’intercepter les erreurs et de renvoyer une réponse adaptée au lieu de laisser l’application planter et rendre le serveur inaccessible.

Pour ce qui est du front-end, il a été développé en VueJS, en utilisant des stores pour gérer les données récupérées depuis notre back-end. Nous considérons ce choix pertinent, car nous avons de nombreuses données à manipuler et les stores offrent un système de cache permettant d’éviter un nombre excessif d’appels API inutiles. De plus, ce choix accélère notre développement, puisque l’ensemble de l’équipe maîtrise déjà plus ou moins VueJS.

4. Représentation en base de donnée

Enfin, la base de données utilisée pour le projet est modélisée selon un schéma relationnel permettant de représenter les liens entre les différentes entités du système. Le schéma comprend notamment les tables correspondant aux attractions et aux spectacles, ainsi que leurs relations éventuelles. Ce schéma est illustré dans le fichier fourni.

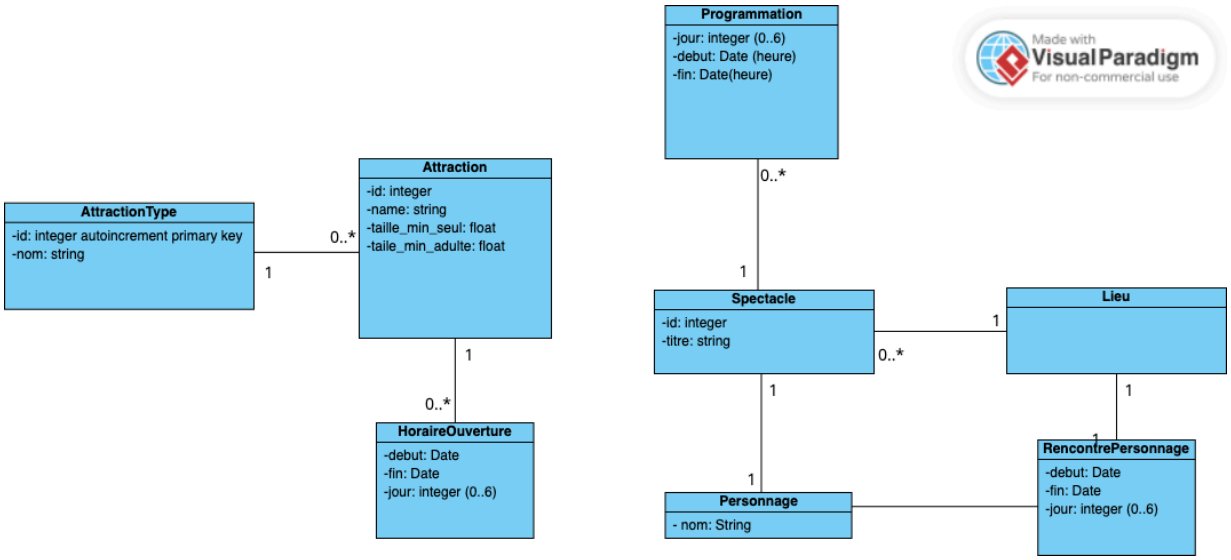

Schéma de la base de données

5. Choix technologiques et justification

Les choix technologiques effectués pour ce projet répondent à des exigences de performance, de maintenabilité et de clarté architecturale.
Le backend a été développé en Java, un langage robuste, mature et largement utilisé pour les applications web professionnelles. Java offre un écosystème riche, particulièrement adapté aux architectures en couches et aux systèmes nécessitant une gestion rigoureuse de la logique métier. L'usage de Maven comme gestionnaire de dépendances permet une gestion structurée du cycle de build, facilite l'ajout de bibliothèques, et garantit la reproductibilité du projet grâce à un fichier `pom.xml` centralisé.

```xml
<dependencies>
    <dependency>
        <groupId>jakarta.servlet</groupId>
        <artifactId>jakarta.servlet-api</artifactId>
        <version>6.1.0</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.glassfish.jersey.containers</groupId>
        <artifactId>jersey-container-servlet</artifactId>
        <version>${jersey.version}</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>
</dependencies>
```

6. Partie front-end

Pour la partie front-end, nous avons opté pour le framework VueJS. Nous avons choisi d'une part ce framework car nous sommes tous habitués dans l'équipe à ce framework et d'autre part car c'est un framework moderne et bien maintenu par la communauté.

Le choix d'utiliser typescript plutôt que javascript est plutôt collectif puisque typescript offre la sécurité des types.

Nous avons utilisé des stores pour stocker et faire nos appels à l'API pour garder une persistance des données et donc optimiser nos rendus de page.

Nos appels API sont interfacé par des fonctions javascript dans un dossier API, mais sont appelés dans nos stores.

Etant donné que nous utilisons TypeScript, nous avons stocké les types de nos objets (Spectacles, Horaire, etc…) dans un dossier à part pour normaliser et faciliter la modification.

Nous avons défini les styles de base (variables, fonts).

Nous avons, bien entendu, pensé à stocker les composants réutilisables dans un dossier.

7. Implémentation des principales fonctionnalités

Côté backend, chaque fonctionnalité métier comme la gestion des attractions ou des spectacles est prise en charge par un controller, un service, puis un DAO.

Le controller expose un endpoint REST, le service applique la logique métier (validation, règles internes), et le DAO interagit avec la base de données pour récupérer ou persister les informations.

Côté front-end nous avons séparé les deux interfaces public et admin dans deux dossiers de vues distincts pour s'y retrouver.

8. Sécurité et gestion des accès

Pour la sécurité, nous avons privilégié la validation des formulaires côté back-end. Le front-end inclut également des validations, mais la logique principale de sécurité et de vérification repose sur le back-end.

Nous appliquons une approche aussi proche que possible du *zero-trust*, dans laquelle le client est continuellement vérifié dans ses actions et ses requêtes.

9. Tests techniques et validation

Pour les tests techniques, nous avons tout d'abord testé les endpoints avec POSTMAN pour vérifier les formulaires et testé les cas extrêmes (cas d'erreur, cas où les valeurs sont absurdes).
Une fois que le front-end a été mis en place nous avons testé avec depuis l'IHM (Interface Homme-Machine) pour double validés les formulaires.

10. Perspectives d'amélioration

Ce projet pourrait être amélioré en ajoutant un système d'authentification (via des jetons ou des sessions) pour vérifier sécuriser les endpoints sensibles (fonctionnalité admin par surtout).

Nous aurions pu aussi utiliser un ORM pour mySQL pour améliorer la maintenabilité du code, car nous avons dû bricoler des petites méthodes pour avoir quelque chose de réutilisable, propre et maintenable.

Nous aurions pu aussi rajouter des tests unitaires (pour le front et le back) pour tester les règles métiers.

Enfin, nous aurions pu utiliser un framework comme Spring Boot pour proposer une base de code maintenable.