Site de billetterie

Ticket to the Moon

Date de mise à jour : 27/06/2022
Version : 1.1

Nom	FENGAROL	Numéro d'auditeur	XXX

Table des matières

1. Objectif du document 3
2. Technologies utilisées 3
2.1. Contraintes techniques 3
2.2. Serveur web 3
2.3. Stockage des données 3
2.4. Couche de persistance 3
2.5. Couche métier 3
2.6. Couche service 3
2.7. Couche présentation 4
2.8. Couche de sécurité et d'authentification 4
2.9. Environnement de développement 4
2.10. Test unitaire 4
2.11. Autres technologies utilisées 4
3. Architecture 5
3.1. Packages et dépendances 5
3.1.1 Backend 5
3.1.2 Frontend 7
3.2. Déploiement 8
4. Cas d'utilisation 9
4.1. Cas d'utilisation d'un gestionnaire 9
4.1.1 Créer un évènement 9
4.1.2 Créer un tarif 14
5. Regroupement des classes 16
5.1. Groupe domaine 17
5.2. Groupe cycle de vie 17
5.3. Groupe service 18
5.4. Groupe interface utilisateur et système 18
6. Annexes 19
6.1. Terminologie 19

1. Objectif du document

L'objectif de ce document est de définir les besoins techniques et de décrire l'architecture technique et applicative ainsi que les technologies choisies pour le projet d'un site de Billetterie en ligne pour des spectacles appelé ***Ticket to the Moon***.

La première partie, sur les choix technologiques identifie les contraintes techniques et abordera les serveurs, le stockage des données, les différentes couches de l'application, l'environnement de développement et enfin les tests unitaires. La seconde partie sur l'architecture présente les packages et dépendances ainsi que la vue de déploiement de l'application. Enfin, selon l'architecture choisie, certains cas d'utilisation spécifiés dans le document d'analyse seront développés de manière plus approfondie.

2. Technologies utilisées

2.1. Contraintes techniques

Dans l'absolu, cette application n'a pas de contraintes techniques. Toutefois, nous nous limiterons ici aux connaissances et/ou à l'intérêt de l'équipe de développement, et cela sous des versions récentes et stables. Ainsi les deux Framework principaux seront SpringBoot et VueJs avec une base de données MySQL le tout déployé sous Docker.

2.2. Serveur web

Le Framework Spring Boot sur lequel repose le backend permet de choisir le serveur d'application de son choix. Par défaut, il embarque le serveur d'application Apache Tomcat qui a fait ses preuves. C'est donc celui qui sera utilisé.

2.3. Stockage des données

Le serveur de base de données sera hébergé par le serveur de base de données relationnelle MySQL. Le driver JDBC prendra en charge la connexion au serveur.

2.4. Couche de persistance

Le Framework Spring Boot fournit un module permettant de gérer la persistance des données (Spring Data JPA), qui utilise l'API JPA (Java Persistence API). Celui-ci sera utilisé pour gérer la persistance des données. Dans ce cadre, les objets persistants sont appelés des entités JPA et annotés avec @Entity. Les classes permettant d'accéder aux données (les DAO) seront des « component » Spring Boot de type Repository (classes qui étendent l'interface JpaRepository).

2.5. Couche métier

On utilisera des objets métiers sérialisables (appelé des objets de transfert de données ou DTO), créés à partir des entités JPA. Ce sont des POJO : Plain Old Java Object ("Java Objet, tout simplement").

2.6. Couche service

La couche service sera décrite via des « component » Spring Boot de type Service (classe annotée avec @Service).

2.7. Couche présentation

La couche de présentation, qui constitue la partie frontend de l'application, sera gérée par le Framework VueJs en REST (REpresentational state transfer). C'est une architecture qui crée et manipule des services web via des requêtes http. Voici son fonctionnement : Le frontend communique avec le backend par l'intermédiaire de requêtes HTTP qu'il envoie, et qui sont réceptionnées par les « component » Spring Boot de type Controller (annotés avec @RestController). Ces derniers, après avoir intercepté les requêtes HTTP, appellent les objets de la couche métier adéquats. Une fois les traitements métiers effectués, les contrôleurs envoient une réponse HTTP au client.

2.8. Couche de sécurité et d'authentification

Les modules Spring Security et Google Sign-In seront utilisés.

2.9. Environnement de développement

Le développement de l'application se fera sous VS Code. Les dépendances sont gérées par Gradle, et le versionning par Git.

2.10. Test unitaire

Spring Boot offre un module de test (Spring Boot Starter Test) qui utilise les librairies JUnit Jupiter, Hamcrest et Mockito. Nous utiliserons donc ce module avec principalement JUnit Jupiter.

2.11. Autres technologies utilisées

Déploiement :

- **Docker** pour la conteneurisation
- **Docker compose**, un outil pour configurer et orchestrer des multi-conteneurs.
- **Kubernetes** pour l'orchestration de conteneurs éventuellement.

Intégration continue :

- **Nexus** qui permet de gérer et centraliser les paquets sur toute la chaîne logistique logicielle
- **Drone** est un serveur d'automatisation qui aide à automatiser les parties du développement logiciel liées au build, aux tests et au déploiement, et facilite l'intégration et la livraison continue.
- **SonarQube** est un logiciel libre permettant de mesurer la qualité du code source en continu.

Documentation :

- **Swagger-ui** est une solution open source qui permet de construire et interagir avec une documentation.

Reverse-proxy :

- **Nginx** est un logiciel libre de serveur Web ainsi qu'un proxy inversé.

3. Architecture

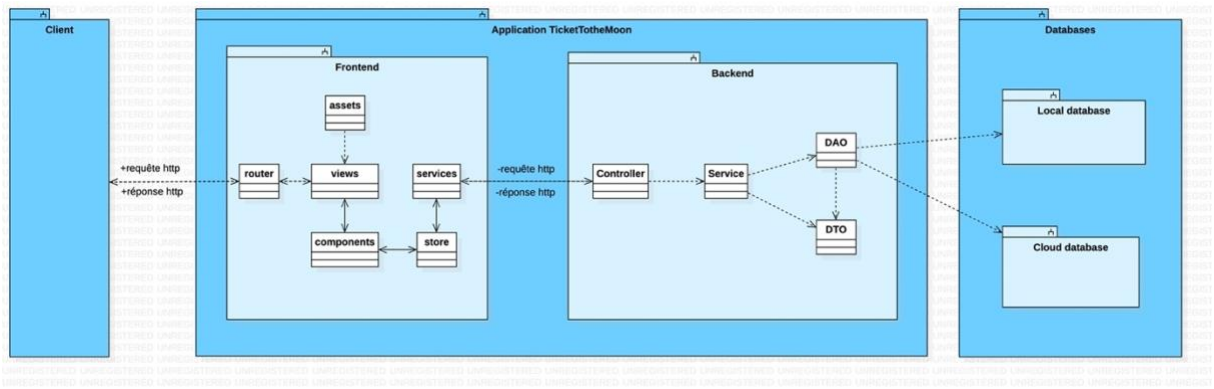
3.1. Packages et dépendances

Une architecture classique à trois couches permet la séparation des tâches, une grande scalabilité, modularité, évolutivité et fluidité. En outre, elle permet à une entreprise qui le voudrait d'avoir son propre front. Voici les couches :

1. La couche présentation avec le frontend
2. La couche applicative avec le backend
3. La couche base de données

Le backend utilise entre autres les design patterns suivants :

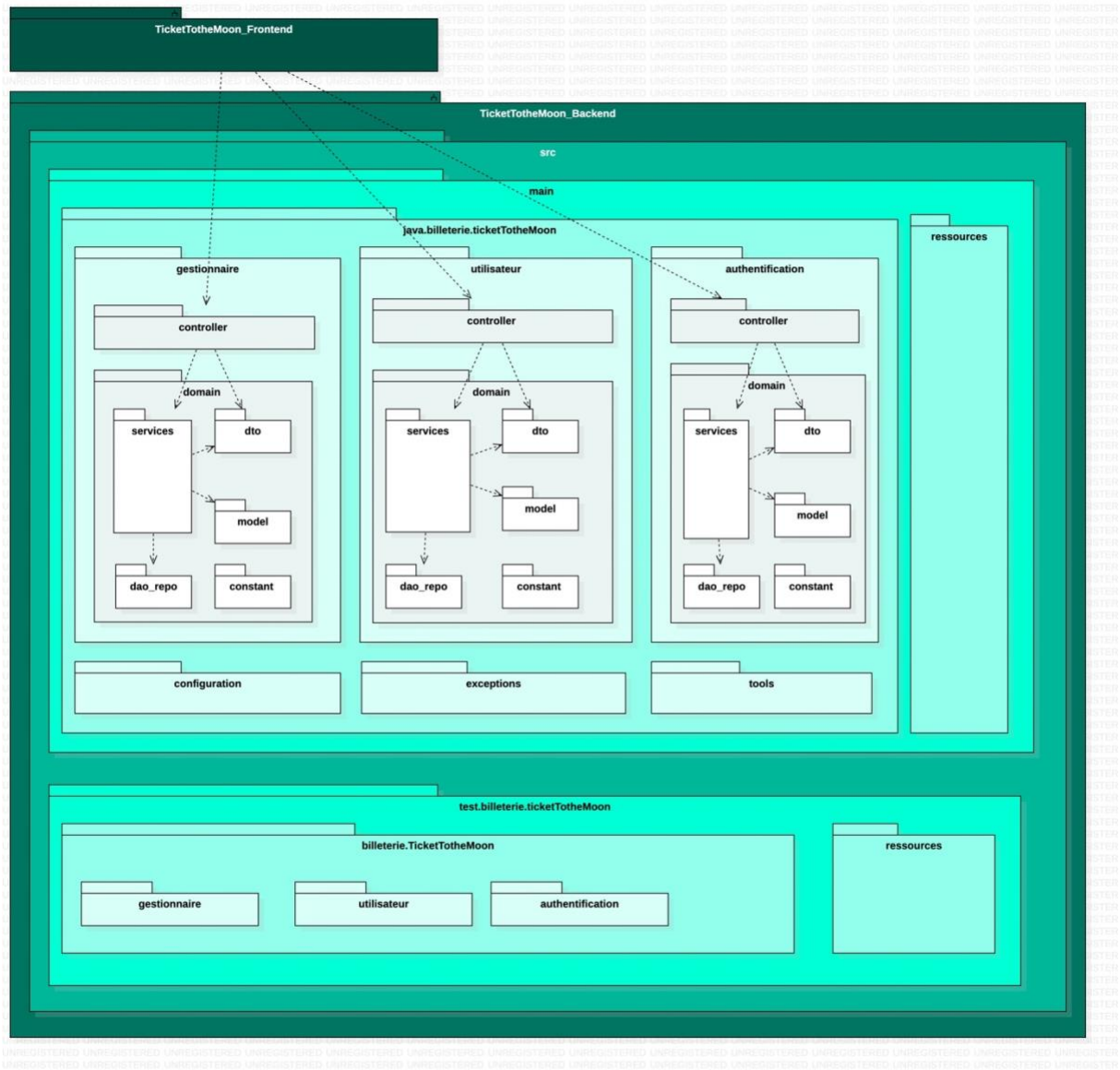
- **MVC** : utilisé pour dissocier les traitements de la visualisation.
- **DAO** : utilisé pour pouvoir se connecter à plusieurs bases de données distinctes.
- **DTO** : utilisé pour transporter plusieurs objets distants en un seul appel distant ou pour les données d'un objet distant non transportable sur le réseau (pas sérialisable).



Idéalement, dans chaque dossier du projet un rapide README.md présente le dossier et les conventions à suivre et les requis.

3.1.1 Backend

Après débat, l'application suit une structure par module (authentification, gestionnaire, utilisateur) puis par couche (controller, service, domain…). C'est une structure un peu lourde mais qui a l'avantage d'être claire et de pouvoir facilement remplacer un module en cas de refactoring.
Les différentes classes sont suffixées du nom de leur fonction (xxxController, xxxService, xxxDto, xxxRepository, xxxException). Les interfaces seront suffixées d'un « able » (ex. : Validable)



- **Fichiers .gradle :** build.gradle pour décrire les dépendances, plugins et repositories. gradle.properties pour les constantes de configuration.
- **main/java/billeterie/ticketTotheMoon :** contient les fichiers sources principaux.
  - **Application.java :** classe principale et l'application, contenant la méthode « main », et point d'entrée de l'application coté backend (annotation @SpringBootApplication)
  - **Configuration** : Contient les classes de configurations du back concernant la sécurité, l'authentification, certaines annotations...
  - **Gestionnaire :** L'ensemble des packages et classes concernant l'utilisateur « gestionnaire » qui gère entre autres les évènements du site de billetterie.
    - **Controller** : Contient les components des classes de type Controller
    - **Domain** : Contient les packages et classes métiers :
      - **Constant** : les constantes de l'application.
      - **DAO_ Repositories** : les classes représentant les entités JPA responsables des opérations en base de données.
      - **DTO** : les classes DTO des objets métiers.
      - **Model** : les objets métiers.

- **Services** : Contient les components des classes de type Service qui sont responsables de la logique métier et servent à mapper les entités en DTO et inversement.
- **Utilisateur** : L'ensemble des packages et classes concernant l'utilisateur « utilisateur », soit le grand public qui utilise le site de billetterie. La structure par couche est identique à Gestionnaire.
- **Authentification :** L'ensemble des packages et classes concernant l'authentification. La structure par couche est identique à Gestionnaire.
- **Exceptions** : Contient les classes d'exceptions métiers.
- **Tools** : Contient les classes utilitaires. Contient les classes utilitaires.

- **main/ressources :** contient les fichiers de ressources
  - **application.properties :** fichier de configuration
- **tests/java/billeterie/ticketTotheMoon :** contient les fichiers pour les tests
  - **ressources :** contient les fichiers de ressources pour les tests :
    - **application.properties :** fichier de configuration pour l'environnement des test
  - **Autres packages :** L'architecture des packages reprends celle du main. Ces packages contiennent le code source des tests.

3.1.2 Frontend

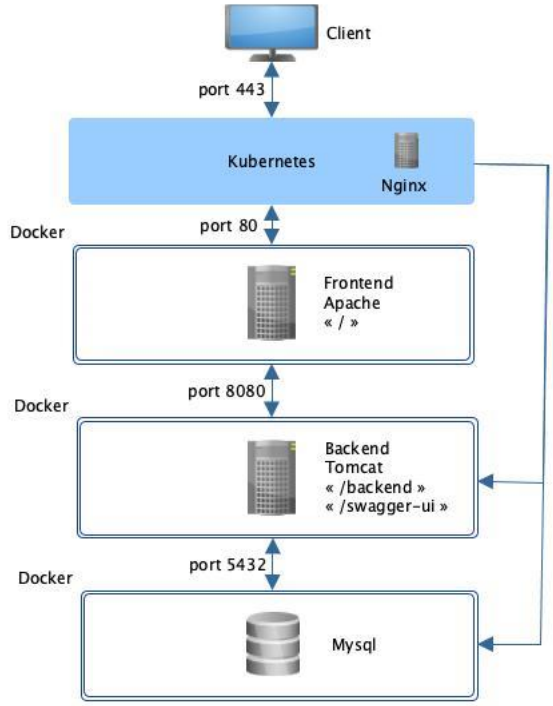
Voici la structure par défaut de VueJs3 augmentée de quelques dossiers pour faciliter la lisibilité sur une application importante.

- **assets** : Répertoire contenant le CSS.
- **components** : Répertoire unique contenant les composants des pages VueJs et respectant les conventions de nommage suivantes :
  - Chaque composant est décrit dans son propre fichier (SFC : single file component).
  - Les composants de base sont nommés BaseXxx.
  - Les composants à usage unique tels que le header ou le footer sont notés par TheXxxx.
  - Les composants à couplage fort tels que les composants enfant et parent seront noté ParentEnfant et Parent. Exemple : CommandeLigneCommande. Commande étant le parent de LigneCommande, l'enfant.
  - Les noms de composants doivent commencer par les mots de niveau le plus élevé (généralement généraux) et se terminer par les mots les plus spécifiques. Exemple : SearchWidgetInput
- **docs :** la documentation.
- **helpers :** fichiers nécessaires à l'ensemble du projet. Exemples : https.js, cache.js, time.js, etc.
- **layouts :** mise en page réutilisable pour l'ensemble du projet
- **mixins :** logique de code réutilisable dans plusieurs composant d'un projet. Par exemple, des objets data, des méthodes, des watchers.
- **plugins :** librairies des tiers (si besoin)
- **router** les conventions pour les pages et leurs routes sont ci-dessous. Le code précisera le nom de la route et de son composant, voici un exemple : <router-link :to="{name: 'UsersIndex'}">Users</router-link>.

Path	Route and Component Name	What it Does
/users	UsersIndex	List all the users
/users/create	UsersCreate	Form to create the user
/users/{id}	UsersShow	Display the users details
/users/{id}/edit	UsersEdit	Form to edit the user

- **store :** un store gère les états partagés de l'application.
- **views :** Répertoire contenant les différentes pages du front (avec l'extension *.vue)
- **3 fichiers : app.vue** défini l'instance root de l'application ; **main.js** crée un contexte et importe l'instance app.vue ; **globals.js** pour les variables globales.
- **services** : Répertoire contenant les services relatifs aux vues, permettant les appels aux API du back.
- **static** : Répertoire contenant le contenu statique tel que les images.

3.2. Déploiement



Pour déployer l'application en architecture 3-tiers, nous utilisons Docker, qui est une plateforme permettant de lancer l'application dans des conteneurs et Kubernetes qui orchestre ces conteneurs. L'application est donc découpée en trois conteneurs :

- Toutes les connexions entrantes seront sur le port 443 du Kubernetes et redirigées en interne via Nginx (reverse proxy) dans Kubernetes.
- Le frontend sur un serveur Apache, exposé en TCP sur le port 80 avec un contextRoot « / » et un volume pour faire persister les données.
- Le backend sur un serveur Tomcat, en TCP sur le port 8080 avec contextRoot « /backend » et un volume pour faire persister les données. Un autre contextRoot sera disponible « /swagger-ui » pour la documentation REST.
- La base de données sera sur un serveur MySQL exposé en TCP sur le port 5432.

4. Cas d'utilisation

A la différence de la phase d'analyse, les objets décrits ici sont ceux qui seront implémentés en prenant en compte les technologies choisies et développées aux précédents paragraphes.
La méthode reste la même que pour le document d'analyse, pour chaque cas d'utilisation :

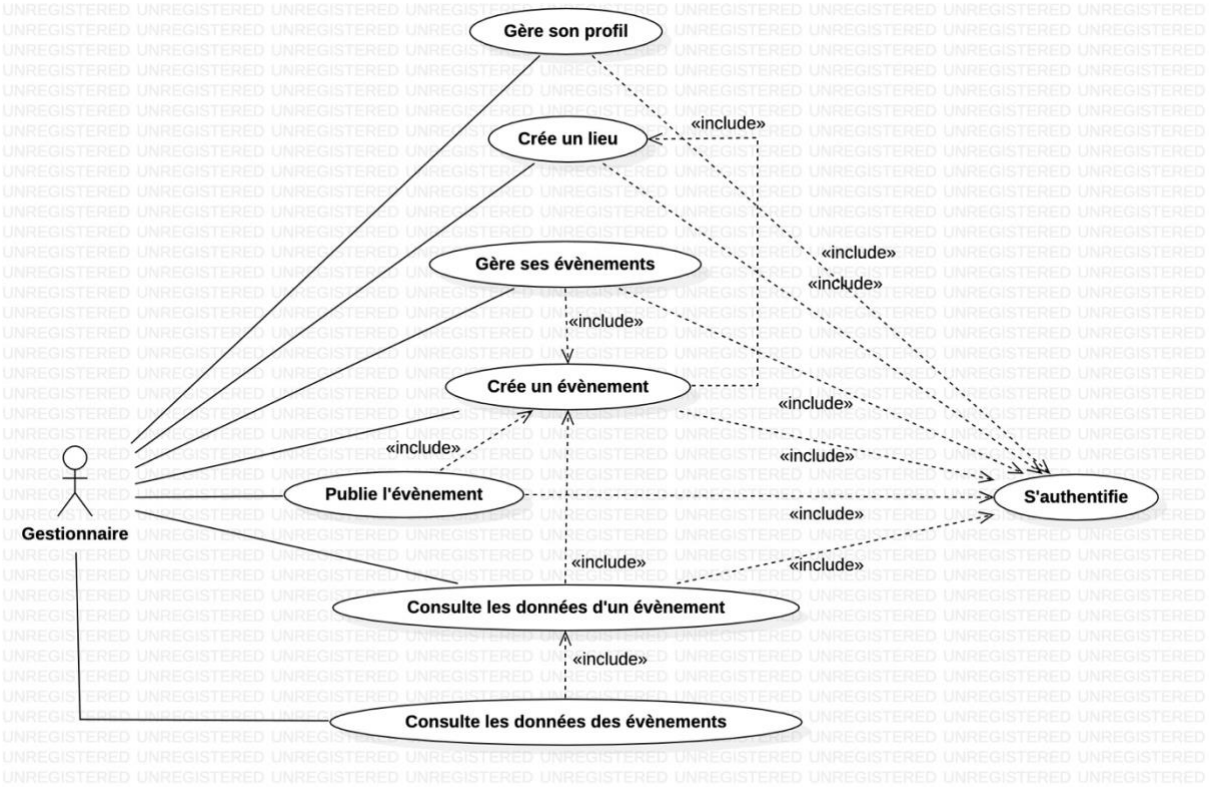
- Les objets sont identifiés et catégorisés par type : <<Entity>>, <<Boundary>>, <<Control>>, <<LifeCycle>> .
- Un diagramme de séquence décrit les interactions entre ces différents objets.
- Un diagramme des classes implémentées résume les liens entre les classes, Leurs attributs et leurs opérations.

Les objets DTO du back ainsi que certaines classes du frontend ne sont pas présentés pour améliorer la lisibilité. Ces objets et fichiers suivent le schéma de l'architecture présentée dans 4.1. Packages et dépendances.

4.1. Cas d'utilisation d'un gestionnaire

Vue macro

Un administrateur doit créer les comptes des gestionnaires. Le gestionnaire doit s'authentifier car toutes ses actions nécessitent une authentification. Il peut gérer son profil, créer et gérer ses évènements, consulter les données d'un évènement ou des évènements dans leur ensemble. Il peut diffuser un évènement ce qui nécessite sa création.

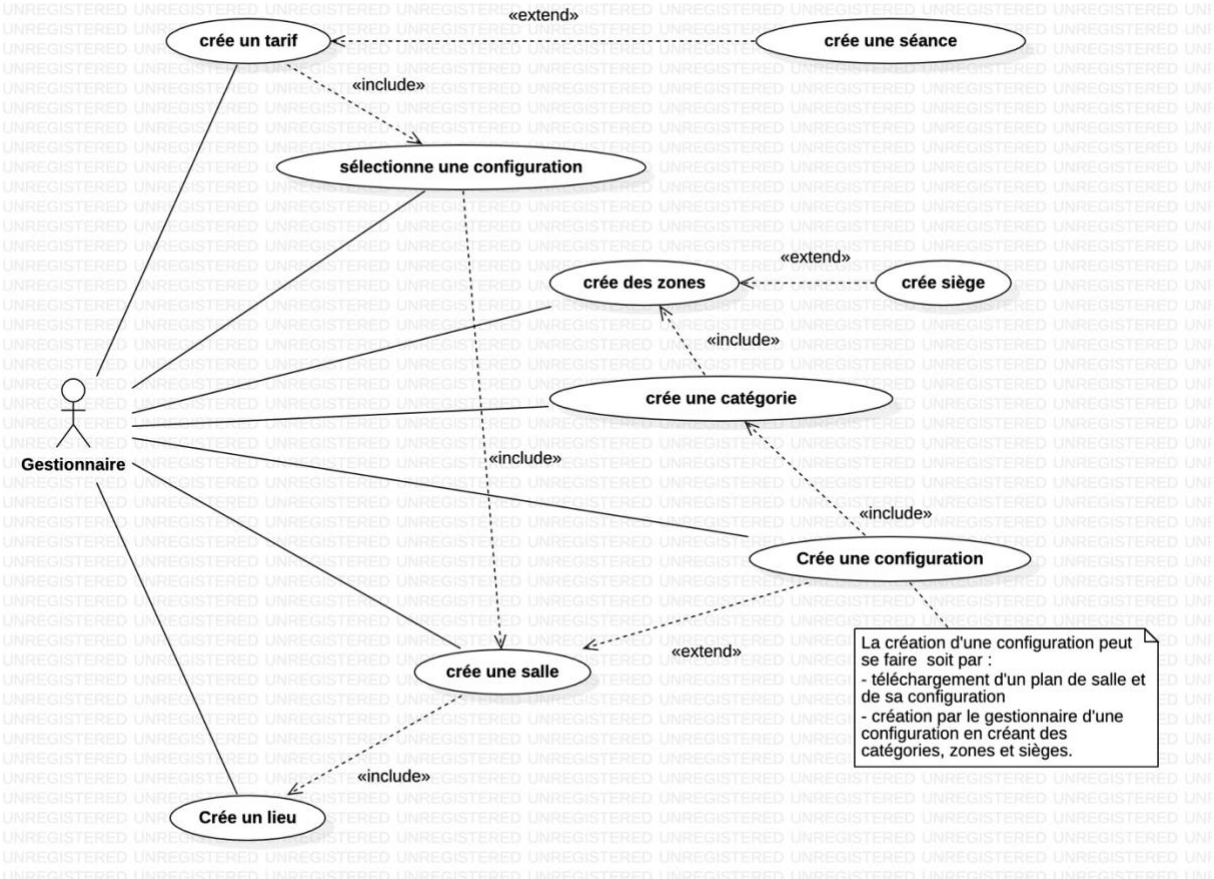


4.1.1 Créer un évènement

C'est le cas d'utilisation le plus important pour le gestionnaire.
Voici les cas d'utilisations inclus dans « crée un évènement ».
Un évènement nécessite la création préalable d'un lieu composé d'une ou de plusieurs salles et d'une configuration de salle comprenant une catégorie (balcon, orchestre...), une zone (zone

A, B, C..), la liste des places numérotées si c'est le cas et un quota de places.
La création d'une configuration n'est pas obligatoire puisque le site ***Ticket to the Moon*** offre la possibilité de télécharger des plans de salles et des configurations toutes faites.
La création d'un tarif nécessite une configuration de salle (téléchargée ou pas) avec au moins une zone (et donc une catégorie). Une catégorie peut avoir 1 ou plusieurs zones et une zone être composée de 1 ou plusieurs catégories. Une zone peut avoir plusieurs prix dans le cas des séances par exemple. Une zone a 0 prix dans le cas des places indisponibles car utilisées par le personnel technique ou car le siège est cassé ou derrière un poteau... Ces places ne seront donc ainsi pas comptabilisées.

Il est à noter pour simplifier et permettre une meilleure lisibilité que :
La création d'un évènement inclus les use case créer une salle et créer une configuration. Le niveau de détail de créer une configuration n'est pas développé ici. Celui-ci nécessite la création de catégories, zones et sièges si nécessaire (Category, Area, Seat). Ces entités sont toutefois disponibles dans la description des classes du backend. La création d'un prix fait aussi l'objet d'un cas d'utilisation à part développé dans le prochain chapitre.



4.1.1.1. Liste des objets candidats

a. Frontend

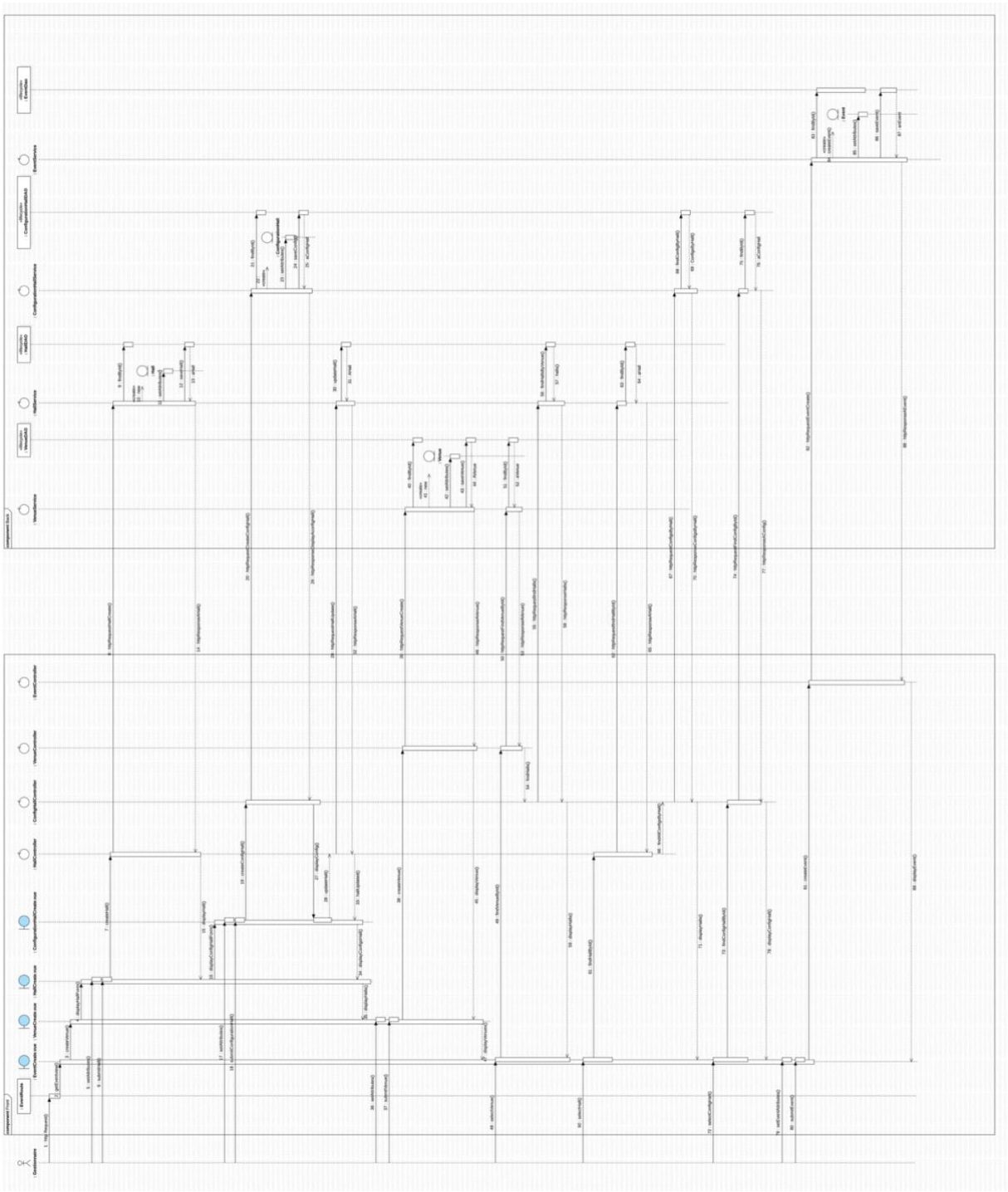
Les objets et fichiers présentés ici sont non exhaustifs en raison du nombre d'objets candidats.
Router :
EventRoute.vue, VenueRoute.vue, ConfigurationRoute.vue
Layout :
EventCreate.vue, EventDisplay.vue,

VenueCreate.vue, VenueDisplay.vue,
HallConfigurationCreate.vue, HallConfigurationDisplay.vue (on y inclut la création ou sélection de la configuration)
Components :
TheHeader.vue, TheFooter.vue,
BaseSaveForm, BaseSendForm,
FormEvent.vue, FormEventVenue.vue, FormEventVenueHall.vue,
FormEventVenueHallConfiguration.vue,
SelectEvent.vue, SelectEventVenue etc…
Assets :
The.css, BaseWidgetButton.css, BaseWidgetInput.css, Form.css, Display.css…
Mixins :
GlobalSecurityVerifyInputMixin.js, clickMixin.js, selectMixin.js, verifyPrice.js, verifyQuota.js
Services :
EventController.vue, VenueController.vue, HallController.vue, ConfigurationController.vue

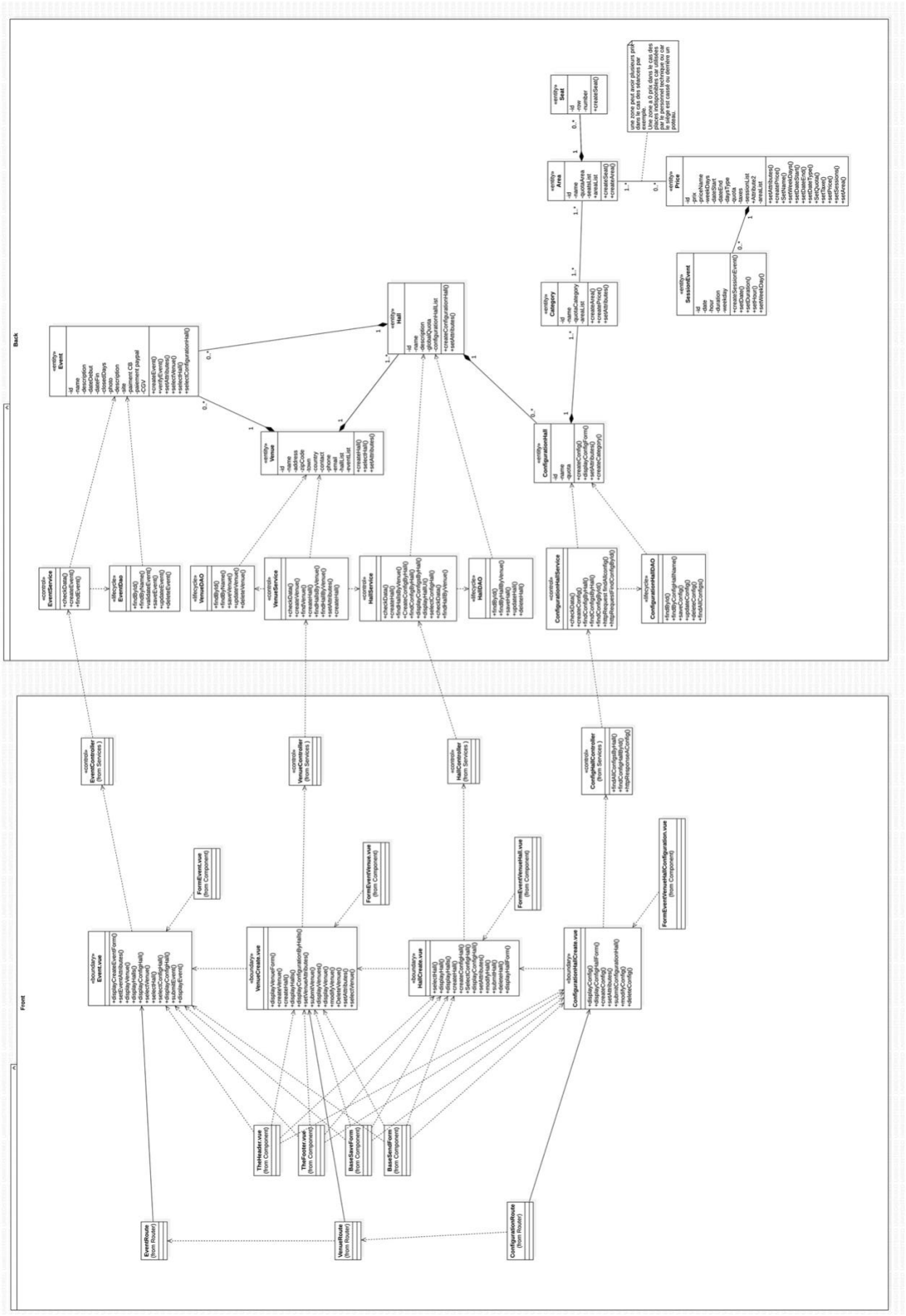
b. Backend

- <<Entity>> :
  - Event : informations de l'évènement
  - Venue : Informations du lieu de l'évènement
  - Hall : Salle où se passe l'évènement
  - ConfigurationHall : Configuration de la salle avec les entités, à titre indicatif, qui en découlent suivantes:
    - Category : une catégorie de place (balcon, orchestre, fosse…)
    - Area : une zone de prix (zone A, zone B, zone C…) dans une catégorie et concernant ou pas des sièges.
    - Seat : la liste des places qui se rapportent à une zone de prix
- <<lifecycle>> : EventDAO, VenueDAO, HallDAO, ConfigurationDAO, CategoryDAO, AreaDAO, SeatDAO
- <<control>> : EventService.java, VenueService.java, HallService.java, ConfigurationService.java, CategoryService.java, AreaService.java, SeatService.java

4.1.1.2. Description des interactions entre objets



4.1.1.3. Description des classes



4.1.2 Créer un tarif

4.1.2.1. Liste des objets candidats

a. Frontend

Les objets et fichiers présentés ici sont non exhaustif en raison du nombre d'objets candidats.
Router :
PriceRoute.vue, SessionRoute.vue,
Layout :
SessionCreate.vue, SessionDisplay.vue
PriceCreate.vue, PriceDisplay.vue,
Components :
TheHeader.vue, TheFooter.vue,
BaseSaveForm, BaseSendForm,
FormEventConfigurationPrice.vue,
FormPriceSession.vue,
Assets :
The.css, BaseWidgetButton.css, BaseWidgetInput.css, Form.css, Display.css…
Mixins :
GlobalSecurityVerifyInputMixin.js, clickMixin.js, selectMixin.js, verifyPrice.js, verifyQuota.js..
Services :
ConfigurationController.vue, AreaController.java, SessionController.vue, PriceController.vue, CategoryController.vue, SeatController.vue

b. Backend

- <<Entity>> :
    - Price : informations d'un prix
    - ConfigurationHall : Configuration de la salle avec les entités qui en découlent suivantes :
        - Category : une catégorie de place (balcon, orchestre, fosse…)
        - Area : une zone de prix (zone A, zone B, zone C…) dans une catégorie de place et concernant ou pas des sièges.
        - Seat : la liste des places qui se rapportent à une zone de prix
        - Session : séance d'un spectacle avec une spécificité de dates, jours, heures.
- <<lifecycle>> : PriceDAO, ConfigurationHallDAO, AreaDAO, SessionDAO CategoryDAO, AreaDAO, SeatDAO, PriceDAO
- <<control>> : ConfigurationService.java, SessionService.java, AreaService.java, PriceService.java, CategoryService.java, SeatService.java

4.1.2.2. Description des interactions entre objets

La création d'un tarif nécessite une configuration de salle (téléchargée ou pas) avec au moins une zone (et donc une catégorie). Une catégorie peut avoir 1 ou plusieurs zones et une zone être composée de 1 ou plusieurs catégories. Une zone peut avoir plusieurs prix dans le cas des séances par exemple. Une zone a 0 prix dans le cas des places indisponibles car utilisées par le personnel technique ou car le siège est cassé ou derrière un poteau… Ces places ne seront donc ainsi pas comptabilisées dans le chiffre d'affaires.

Étapes de création sans séances :

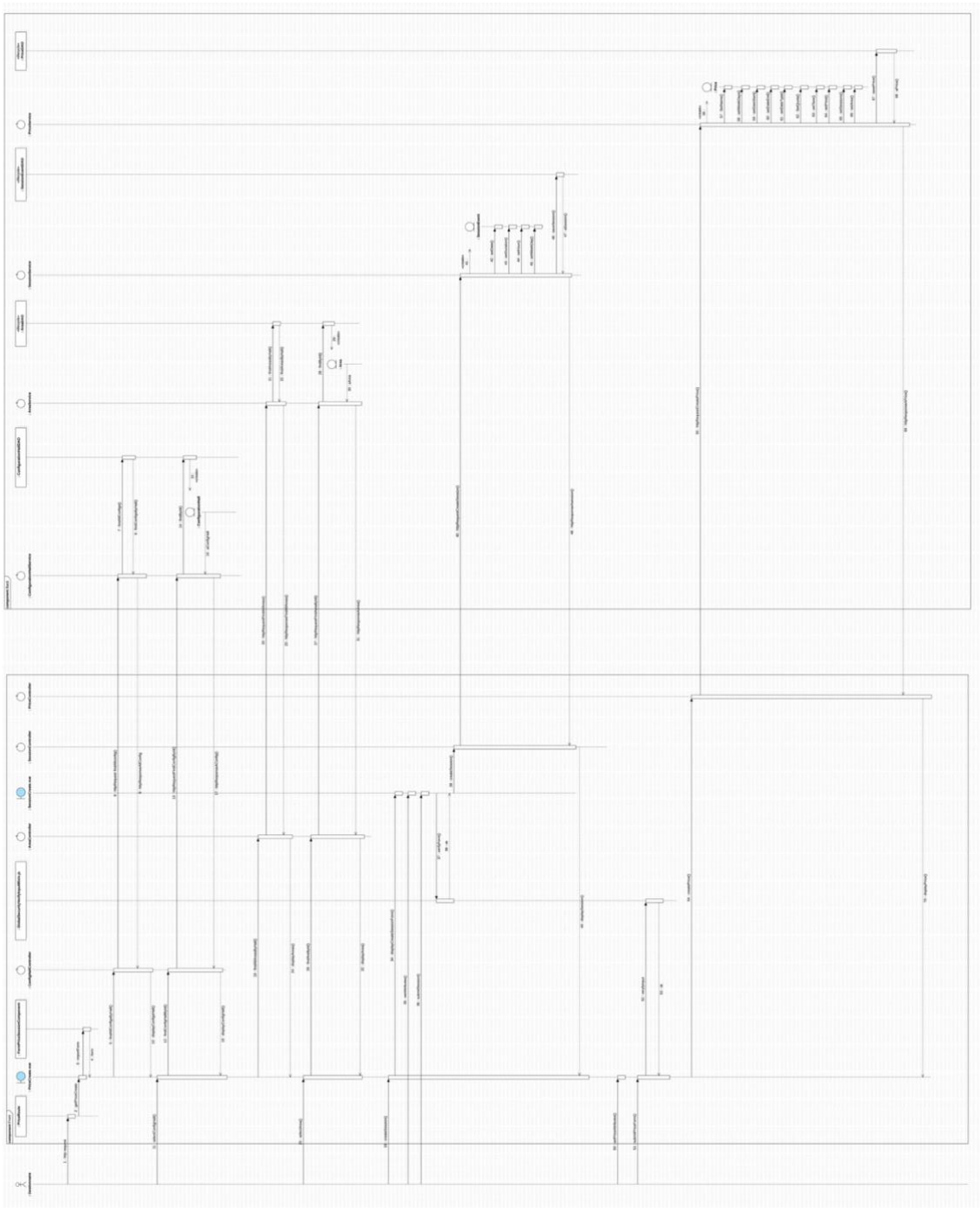
- Sélection d'une configuration pour l'évènement

- Sélection d'une zone : zone A
- Définir prix, nom

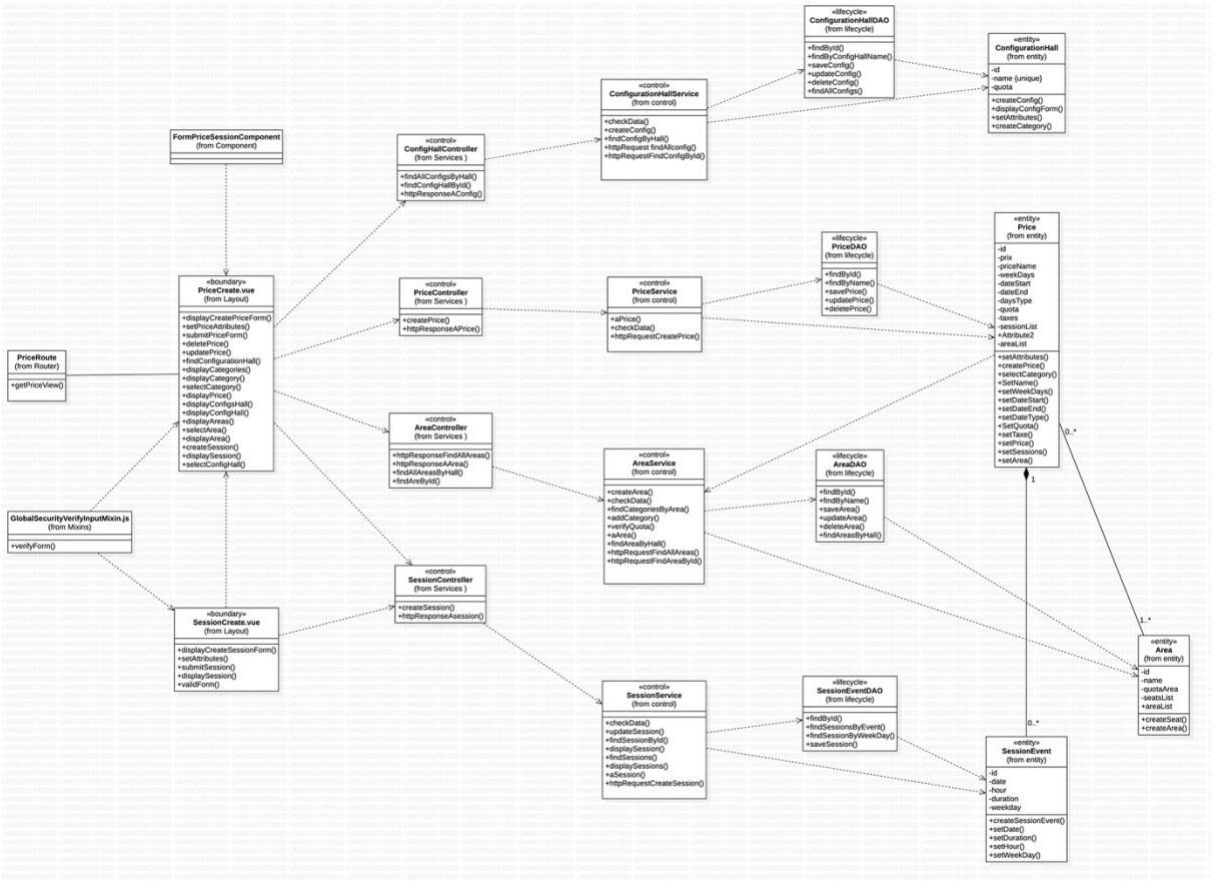
Étapes de création avec séances :

- Sélection d'une configuration pour l'évènement
- Sélection d'une zone : zone A
- Création d'une ou de plusieurs séances
- Définir prix, nom

C'est ce dernier cas que nous développerons ici.



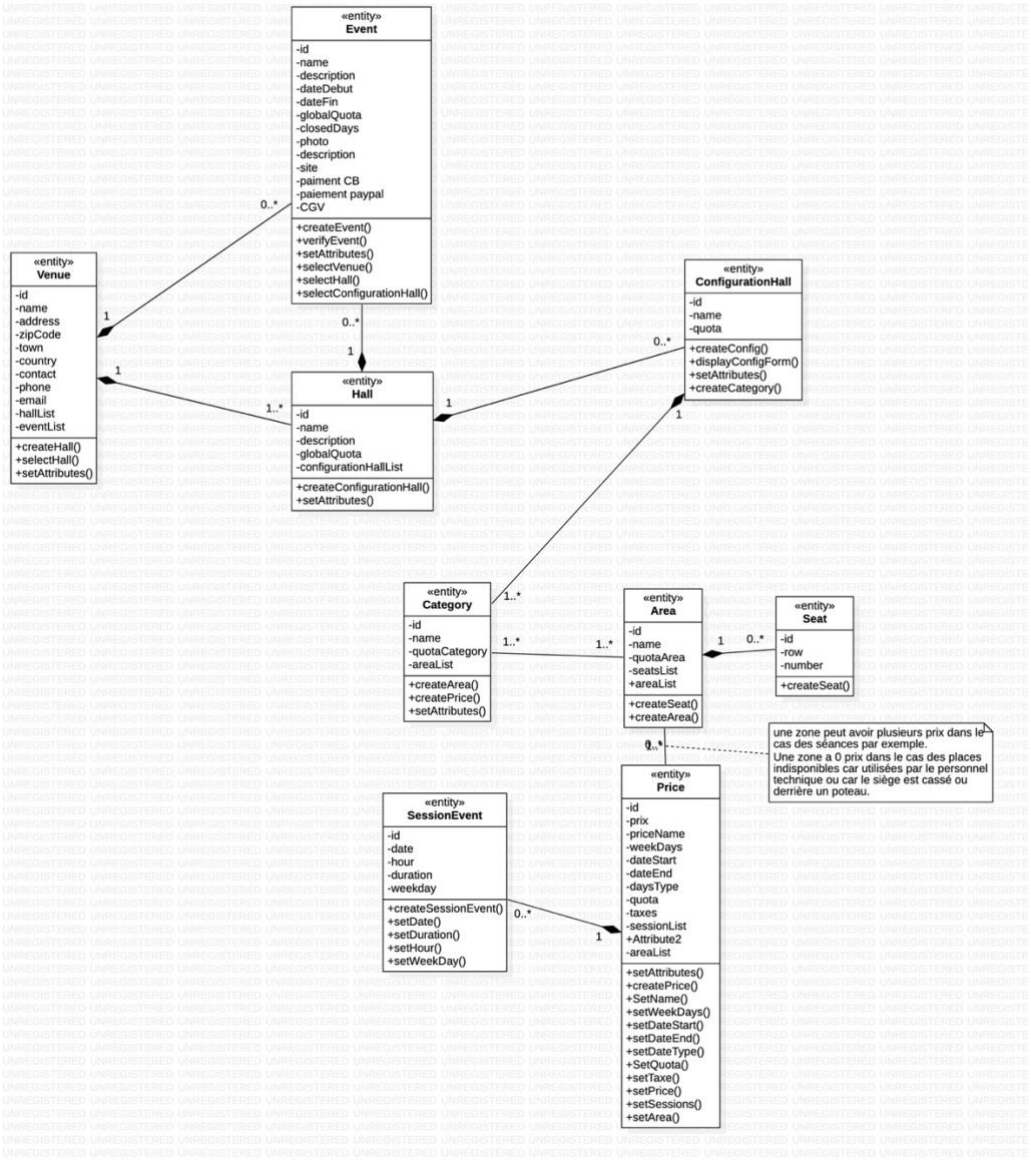
4.1.2.3. Description des classes



5. Regroupement des classes

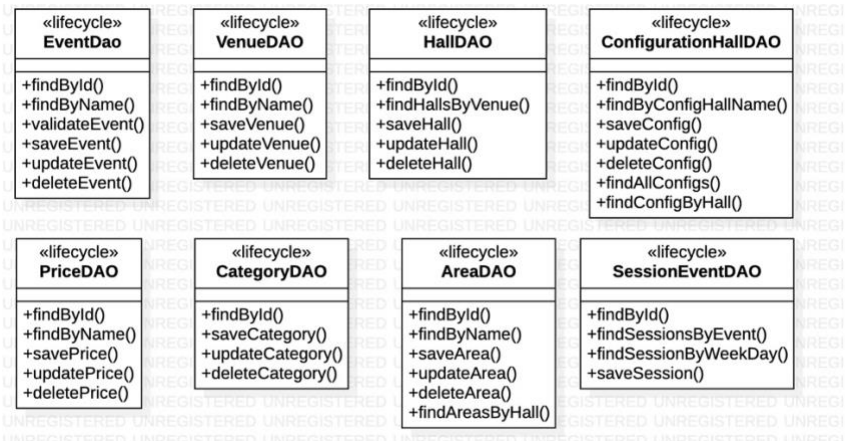
Les attributs et opérations ne sont pas exhaustifs et peuvent être sujet à changement d'appellation lors de la phase développement. Seules les classes pertinentes sont représentées, l'ensemble de l'architecture est disponible dans le schéma du chapitre 4.1. Packages et dépendances.

5.1. Groupe domaine



5.2. Groupe cycle de vie

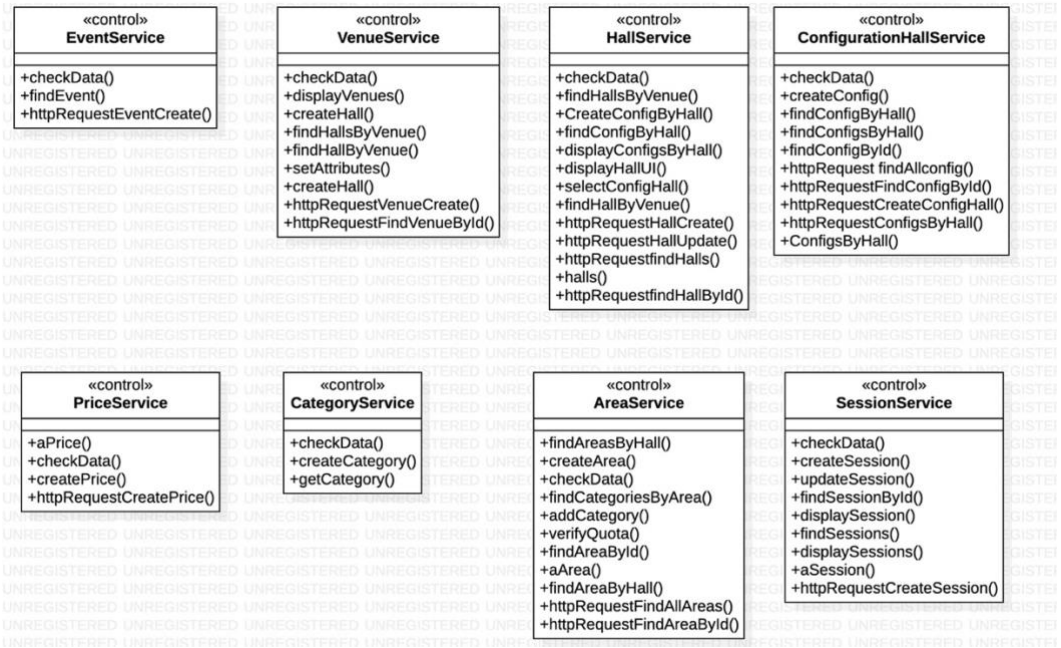
a. Backend



5.3. Groupe service

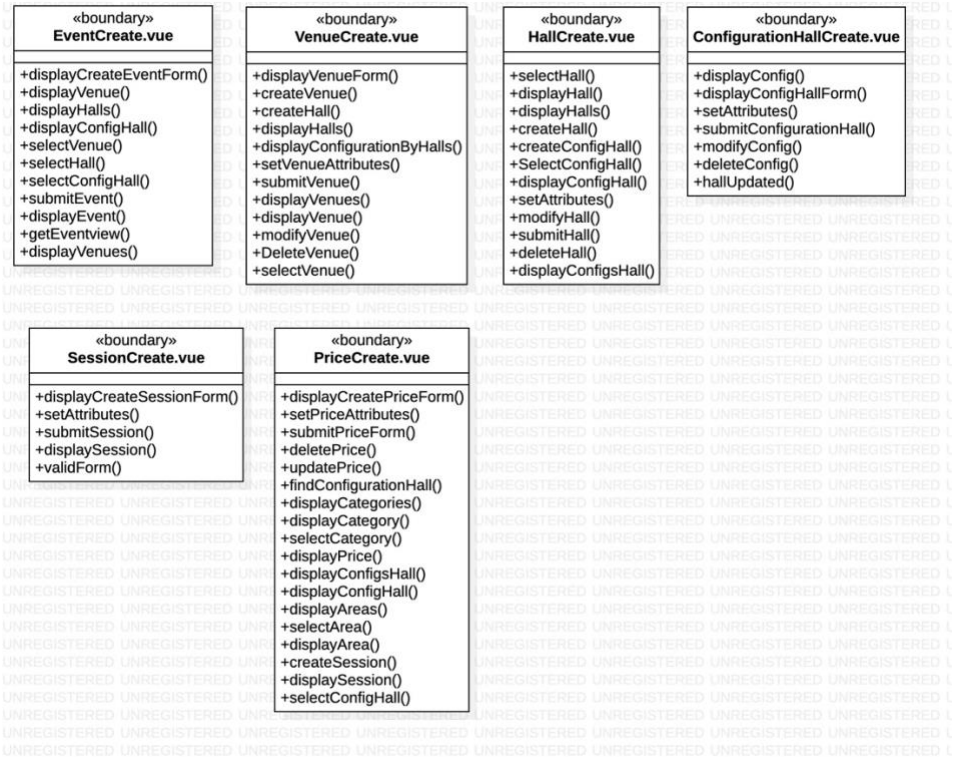
b. Backend

Pour la lisibilité des schémas précédents, quelques opérations ont été nommées httpRequestXXX.



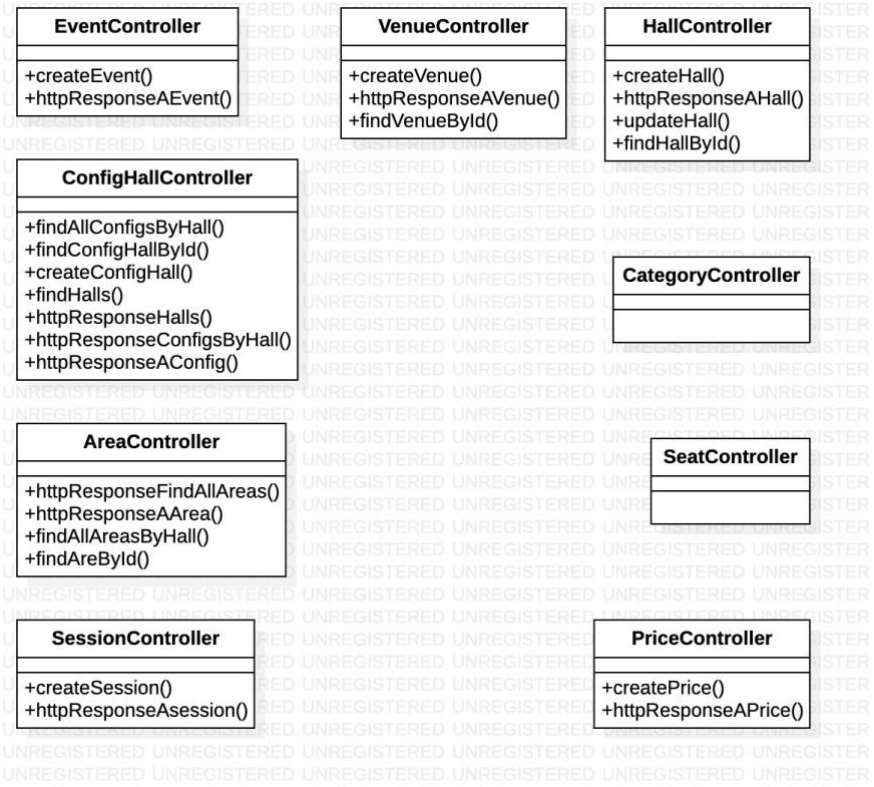
5.4. Groupe interface utilisateur et système

a. Frontend boundary view



a. Frontend boundary controller

Pour la lisibilité des schémas précédents, quelques opérations ont été nommées httpResponseXXX.



6. Annexes

6.1. Terminologie

ENTITY	Désigne les objets métier de notre application, des informations durables et persistantes.
BOUNDARY	Objets qui servent à modéliser les interactions entre le système et ses acteurs.
CONTROL	Objets utilisés pour représenter la coordination, l'enchaînement, la gestion et le contrôle d'autres objets. Façade entre les objets Boundary et Entity.
LIFECYCLE	Objets responsables de trouver les objets Entity et permettant la liaison à un SGBDR
DAO	Data Access Object Classes qui servent à accéder aux informations durables et souvent persistantes
UI	User Interface Classes permettant d'interagir avec les acteurs externes
DTO	Data Transfer Object Patron de conception permettant de représenter un objet métier