Projet Spring - VueJs

Site de billetterie en ligne
appelé “Event Hub”

Virginie Fengarol

Cnam GLG204

EventHub

EventHub est une application de billetterie destinée aux professionnels du secteur événementiel et au grand public.

Objectif de l'application : L'objectif principal d'EventHub est de faciliter la gestion et la vente de billets pour des événements, en proposant une interface pour les gestionnaires de salles et les utilisateurs finaux. D'un côté, les professionnels peuvent gérer les événements, salles, séances et tarifs. De l'autre, le grand public peut visualiser les spectacles, effectuer des réservations et acheter des billets.

Architecture technique de l'application à 3 couches :

- **Couche de présentation (front-office) :** Représentée ici par **event-hub-site**, qui est l'interface utilisateur développée en Vue.js C'est la partie visible par les utilisateurs (public et gestionnaires) pour interagir avec l'application.
- **Couche logique (back-office) :** Représentée par **event-hub-server**, qui est l'API REST écrit en java/Springboot. Cette couche contient la logique métier, c'est-à-dire le traitement des données, la gestion des événements, des salles, des réservations, etc…
- **Couche de données :** En développement l'application dispose d'une base de donnée h2 et des données tests sous flyway.

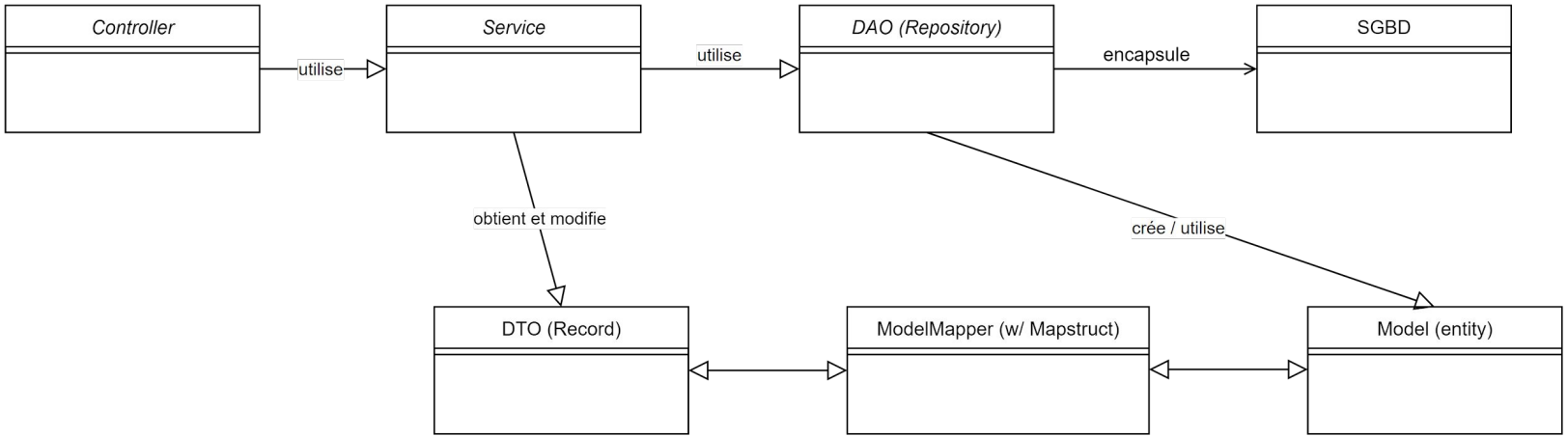
Plan

- Spring
  - Architecture et nouveaux outils : Records et Mapstruct
  - Gestion des erreurs centralisé
  - Testing
  - Documentation : Spring REST Docs et HalExplorer
  - Observabilité et Métrique : Actuator et Gradle buildscan
  - Retour d'expérience
  - Sources utiles
- VueJs
  - Présentation des nouveaux outils
  - Architecture
  - Retour d'expérience
  - Sources utiles
- Architecture finale

Spring

Architecture et nouveaux outils

Architecture



Java Records pour les DTO

Rappel sur le design pattern du "DTO" ("Data Transfer Object") : Le DTO permet de découpler la logique métier de la présentation. Son but est de transférer uniquement les données nécessaires, sans logique métier, pour réduire le volume d'informations échangées. Cela permet de diminuer le trafic réseau et d'améliorer les performances. Les DTO sont souvent utilisés pour encapsuler des données provenant de plusieurs entités ou objets métier, afin de les transmettre de manière efficace et structurée. Par exemple, une facture et ses lignes d'articles, taxes, etc., peuvent être réduites à l'essentiel pour l'affichage : numéro de facture, date, client, montant total.

Au lieux d'utiliser des classes POJO (Plain Old Java Objects) pour créer des DTO, Java propose un nouveau type de classe introduit en standard en Java 16 : les **RECORDS**. Il est principalement utilisé pour créer des classes immuables dont le seul but est de stocker des données.

Caractéristiques :

- Une déclaration de record spécifie dans un en-tête une description de son contenu
- les méthodes d'accès appropriées, le constructeur, les méthodes equals, hashCode et toString sont créées automatiquement.
- Les champs d'un enregistrement sont finaux car la classe est conçue pour servir de simple "porteur de données"

Cette solution remplace Lombok qui est une bibliothèque externe et qui a le désavantage de ne pas fonctionner pour l'analyse de la couverture du code. Elle peut néanmoins restée utile pour les entity et être couplé aux Records si besoin.

Code mise en place et utilisation

```java
@Entity
public class Employee implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;

    public Employee() {}

    public Employee(Long id, String username, String password) {

        setId(id);
        setUsername(username);
        setPassword(password);
    }

    public Employee(String username, String password) {
        setUsername(username);
        setPassword(password);
    }
}
```

```java
public record EmployeeDTO(Long id, String username, String
password) {}
```


Génère les getters, setters et les méthods equals(), hashCode() et toString().

Pour accéder à un champs du DTO :

```java
String username = employeeDTO.username()
```

Mapstruct pour le mappage

MapStruct automatise le mappage entre les entity et les DTO en utilisant les annotations. Il génère des mappages de beans au moment de la compilation, ce qui garantit des performances élevées.

Contrairement à ModelMapper, une autre solution de mappage, MapStruct est compatible avec les Records de Java. D'autre part, MapStruct se révèle plus simple d'utilisation, configuration et mise en place.

Code mise en place

```
@Entity
public class CarDto {
      private String make;
      private int seatCount;
      private String type;
      //constructor, getters, setters etc.
}
```

```
public record CarDto (String make, int
seatCount, String type){}
```

```
@Mapper
public interface CarMapper {

   CarMapper INSTANCE = Mappers.getMapper(
CarMapper.class );

   @Mapping(source = "numberOfSeats", target =
"seatCount")
   CarDto carToCarDto(Car car);
   Car carDTOToCar(CarDTO carDTO);

   @IterableMapping(elementTargetType = CarDTO.class)
   List<CarDTO> ToCarDtos(Iterable<Car> cars);

   @IterableMapping(elementTargetType = Car.class)
   List<Car> ToCars(Iterable<CarDTO> carDTOs);
}
```

Code utilisation

```java
@Test
public void shouldMapCarToDto() {
    //given
    Car car = new Car("Morris", 5, CarType.SEDAN);
    //when
    CarDto carDto = CarMapper.INSTANCE.carToCarDto(car);
    //then
    assertThat(carDto).isNotNull();
    assertThat(carDto.getMake()).isEqualTo("Morris");
    assertThat(carDto.getSeatCount()).isEqualTo(5);
    assertThat(carDto.getType()).isEqualTo("SEDAN");
}
```

```java
@Service
public class CarService() {

    private CarRepository carRepository;

    @Autowired
    public CarService(CarRepository carRepository) {
        this.carRepository = carRepository;
    }

     public List<CarDTO> findAll(){
        Iterable<Car> cars = carRepository.findAll();
        // Mapping des propriétés entre Car et CarDTO
        List<CarDTO> carDTOs = CarMapper.INSTANCE.toCarDTOs(cars);
        return carDTOs;
    }
}
```

Gestion des erreurs centralisé

Un controller

Spring propose l'annotation de class **@ControllerAdvice** pour centraliser la gestion des exceptions pour l'ensemble des controllers. Cette annotation agit comme un intercepteur des requêtes entrantes et des réponses sortantes des contrôleurs sur les annotations de type **@RequestMapping**, facilitant ainsi la gestion des exceptions au sein de l'application.

Une mise en place simple : À l'intérieur de la classe annotée avec @ControllerAdvice, des méthodes sont annotées avec @ExceptionHandler pour gérer des exceptions spécifiques. Ces méthodes seront invoquées lorsque l'exception correspondante est levée dans l'un de vos contrôleurs. D'autres annotations sont disponibles pour gérer les codes d'erreur.

Code exemple

```
@ControllerAdvice
public class GlobalExceptionHandler {

        @ExceptionHandler(CreateException.class)
        @ResponseStatus(HttpStatus.BAD_REQUEST)
        @ResponseBody
        public ResponseEntity<GlobalErrorResponse> handleCreateException(CreateException exception,
        HttpServletRequest request) {
                GlobalErrorResponse errorResponse = new GlobalErrorResponse(ZonedDateTime.now(),
                HttpStatus.BAD_REQUEST.value(), request.getRequestURI(),
                "A create object error occurred : " + exception.getMessage());
                return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
        }
}
```

Testing

Testing

Dans ce projet j'ai utilisé plusieurs type de test :

- **MockMvc** permet de tester les applications Spring MVC en simulant des requêtes et des réponses HTTP sans démarrer de serveur. Il prend en charge toute la gestion des requêtes via des objets fictifs. J'ai couplé cette solution à **Spring Rest Docs** pour générer automatiquement la documentation des API à partir des **tests unitaires ou d'intégration partiels**.
- **TestRestTemplate** simule de véritables interactions client-serveur. Il envoie des requêtes HTTP réelles à une instance de l'application qui s'exécute sur un port aléatoire pendant les tests. C'est idéal pour les **tests d'intégration**, car il recrée des conditions proches de la production.
- **@WebMvcTest** est utilisé pour tester la couche web de l'application, en isolant les contrôleurs. Les autres composants (services, repositories) ne sont pas chargés, ce qui permet des **tests unitaires** plus rapides et spécifiques aux endpoints.

Documentation

Spring REST Docs

Spring REST Docs est un projet open source au sein de l'écosystème Spring destiné à générer la documentation des API RESTful en utilisant des tests automatisés. Il combine une documentation manuscrite écrite avec Asciidoctor et des extraits de code générés automatiquement par le résultat des tests avec Spring MVC Test ou autre.

Sa mise en place nécessite Java 17, Spring Framework 6, de la configuration de dépendance ainsi que la rédaction des tests (MockMVC ou un autre). Par défaut, la documentation sera généré dans le dossier `**build/generated-snippets**` pour Gradle.

Pour générer la documentation faire simplement `**gradle clean build**`.

Code

```
@ExtendWith({RestDocumentationExtension.class, SpringExtension.class})
@SpringBootTest
public class EmployeeControllerTest {
  private MockMvc mockMvc;
  @Autowired
  private ObjectMapper objectMapper;
  String url = "http://localhost:8080/api/";
  @Autowired
  private EmployeeRepository employeeRepository;

  @BeforeEach
  public void setUp(WebApplicationContext webApplicationContext,
          RestDocumentationContextProvider restDocumentation) {
      this.mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext)
              .apply(documentationConfiguration(restDocumentation))
              .alwaysDo(document("{method-name}",
                      preprocessRequest(prettyPrint()),
                      preprocessResponse(prettyPrint())))
              .build();
  }
```

```
@Test
public void employeesCreateExample() throws Exception {
    Map<String, Object> employee = new HashMap<>();
    employee.put("username", "username1");
    employee.put("password", "secretpwD%1");
    employee.put("email", "mymail@mail.fr");

    String employees = this.mockMvc
            .perform(post(url + "employees").contentType(MediaTypes.HAL_JSON)
                    .content(this.objectMapper
                            .writeValueAsString(employee)))
            .andExpect(status().isCreated()).andReturn().getResponse()
            .getHeader("Location");
  }
}
```

Génère les snippets

Par défaut, six snippets sont générés :

- `<output-directory>/index/curl-request.adoc`
- `<output-directory>/index/http-request.adoc`
- `<output-directory>/index/http-response.adoc`
- `<output-directory>/index/httpie-request.adoc`
- `<output-directory>/index/request-body.adoc`
- `<output-directory>/index/response-body.adoc`

Le fichier `src/docs/asciidoc/api-guide.adoc` peut être personnalisé :

```
== My doc

=== Get all employees

.request

include::{snippets}/employees-create-example/curl-request.adoc[]
```

Hal Explorer

HAL (Hypertext Application Language) est un format de représentation de ressources dans les services web RESTful, conçu pour faciliter la navigation entre les ressources et améliorer la découvrabilité des API.

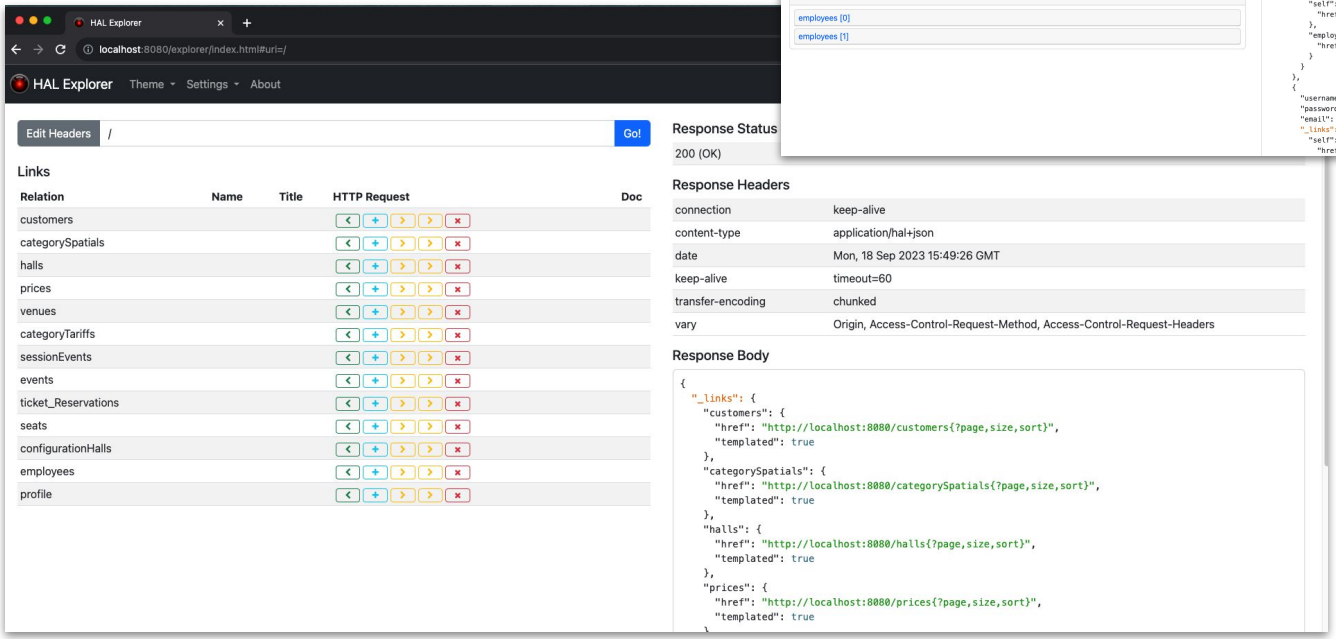
Hal Explorer est composé de :

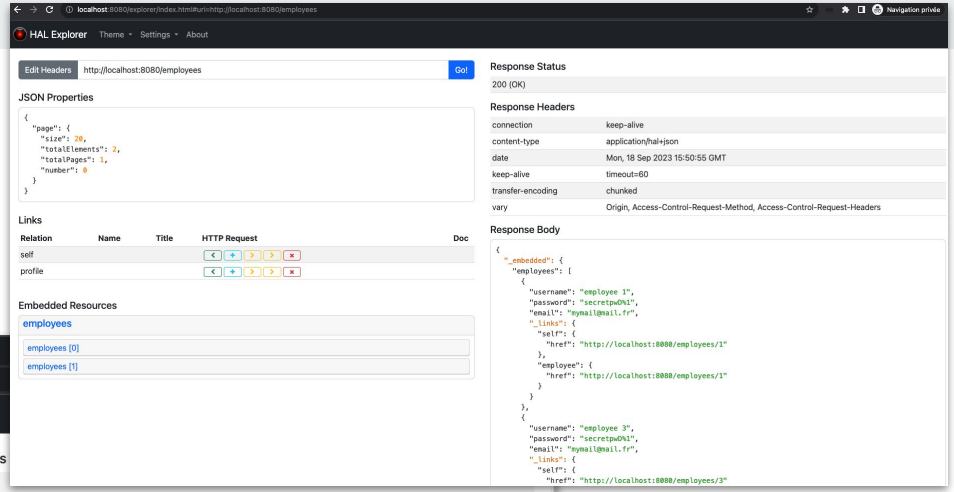
- **HAL Browser**, une interface utilisateur basée sur le navigateur pour explorer des API basées sur HAL. Il permet aux développeurs de visualiser, naviguer et tester facilement les API à travers les ressources exposées.
- **HAL-FORMS** qui introduit une manière standard de représenter les formulaires web dans les réponses API basées sur HAL. Il utilise des éléments HAL spécifiques pour définir les champs de formulaire, les actions, les méthodes HTTP, les URI cibles, etc.

Sa mise en place est très simple puisqu'elle ne nécessite que l'ajout de sa dépendance à l'outil de build. Cet outil permet de tester et contrôler rapidement son API lors du développement.

Son fonctionnement : Hal explorer exécute `**curl -H "Accept: application/hal+json" http://localhost:8080/**` et récupère la réponse JSON contenant des liens hypermédia (_links) et les analyse pour permettre une navigation dynamique à travers les différents endpoints de l'API.

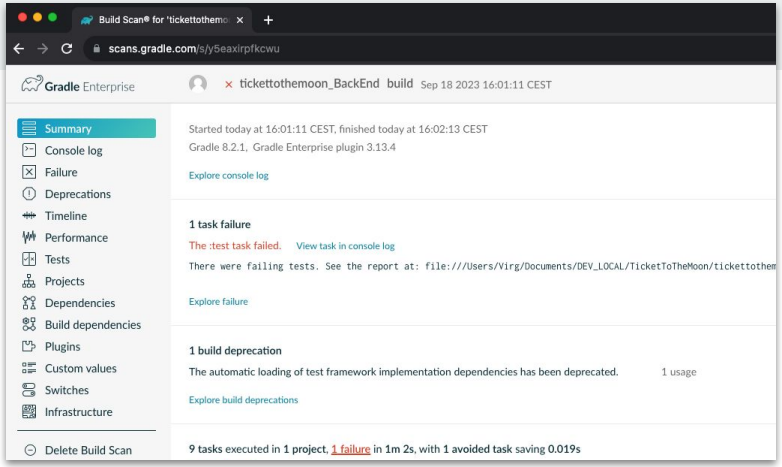
Copie d'écran





Observabilité et Métrique

Outils utiles



Gradle Build Scan `gradle build - -scan`

Outil avancé de suivi de build. Il permet de capturer des métriques, des journaux et des informations sur les dépendances pour faciliter l'analyse des performances et la résolution des problèmes. Il offre aussi une gestion efficace du cache, ce qui permet de stocker les artefacts de construction dans le cloud, réduisant ainsi le temps de construction et les ressources nécessaires, tout en garantissant la cohérence et la fiabilité des dépendances pour les projets distribués.

Actuator `http://localhost:8080/actuator`

Spring Boot Actuator est un module de Spring Boot qui offre des endpoints pour la surveillance et la gestion des applications, permettant aux développeurs de collecter des métriques, de vérifier la santé de l'application et d'effectuer diverses opérations de gestion.

Retour d’expérience

Cette architecture m'a demandé pas mal de recherche mais je suis vraiment heureuse du résultat qui semble simplifier le développement. C'est en complexifiant le projet que je pourrais toutefois la tester.

- **ce qui fonctionne :**
  - Tout ce qui est décrit dans la documentation disponible http://localhost:8080/api c'est à dire la quasi totalité du programme.
- **ce qui vous a posé un problème:**
  - La conception de la base de données et les relations JPA et notamment la clé composite.
  - Mapstruct : j'ai mis du temps à comprendre le fonctionnement dans les cas plus complexes (ex. : ConfigurationHallMapper, EmployeeMapper , TicketReservationMapper).
  - Rest-Docs: la description des champs est un peu lourde j'ai donc créé des classes utilitaires qui génère dynamiquement des champs (cf. : EntitiesFieldDescriptor, EntitiesFieldDescriptorVenue, VenueControllerTest)
  - Voir le wiki sur github pour les bugs spécifiques et leurs résolutions.
  - J'ai travaillé qu'avec les dernières versions des technologies et donc des bugs non résolus ou qui ont été résolu pendant le projet
  - Le nommage est une chose peu aisé en général, la logique des routes api…
  - La configuration chronophage des outils, du poste de travail, des logiciels pour lequel il faut s'autoformer.

- **ce qui serait intéressant d'ajouter :**
  - Spring :
    - La gestion des erreurs est à revoir selon la norme RFC7807, j'ai vu une conférence sur le sujet… Je ne suis pas heureuse de ma gestion des messages d'erreur
    - Ajouter Spring Security
    - Faire une abstract class pour les services, utiliser les enum pour les entités du style payment_status, category etc..
  - Environnement :
    - SpringBoot-testContainer: une base de donnée test en container. Plus pratique que h2.
    - Déploiement dans le cloud, j'ai commencé à travailler dessus.

Sources utiles

Excellente conférence : https://www.youtube.com/watch?v=IgmeFeTU1a4&t=2648s

Records : https://docs.oracle.com/en/java/javase/17/language/records.html#GUID-6699E26F-4A9B-4393-A08B-1E47D4B2D263

MapStruct : https://mapstruct.org/

Hal Explorer : https://toedter.github.io/hal-explorer/release/reference-doc/

Spring REST Docs : https://docs.spring.io/spring-restdocs/docs/current/reference/htmlsingle/

Gradle buildScan : https://scans.gradle.com/

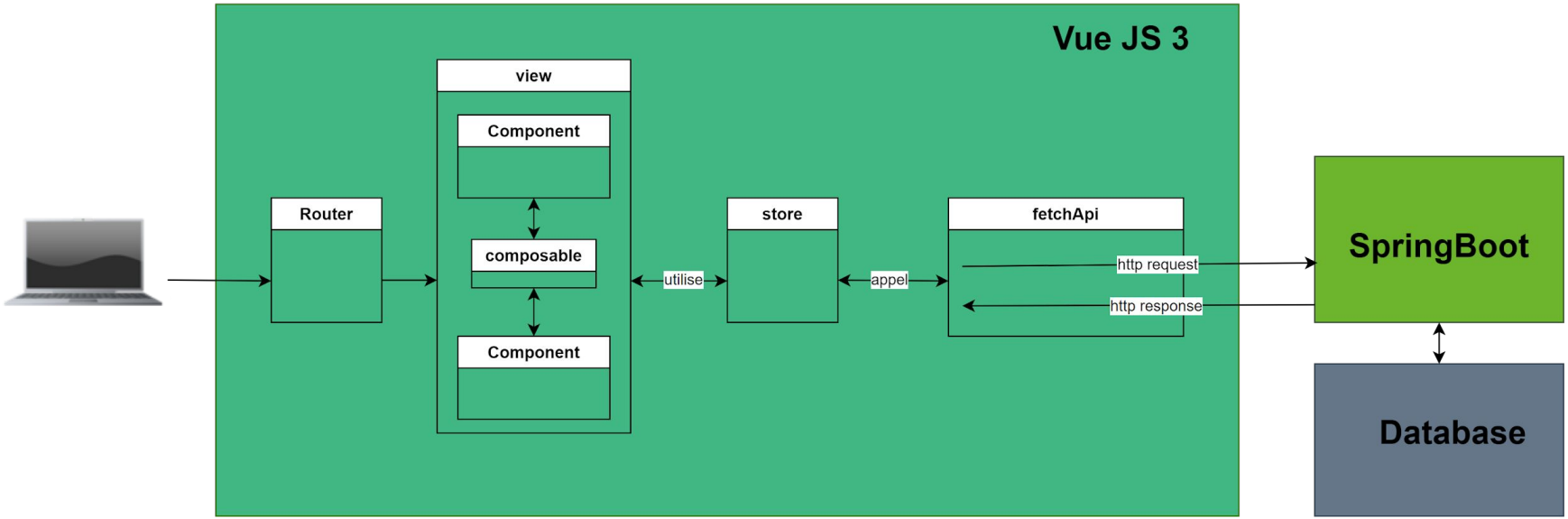
Actuator : https://spring.io/guides/gs/actuator-service/

Vue Js 3

Présentation des nouveaux outils

- **Vite** est outil de développement rapide pour les applications web basées sur JavaScript et TypeScript. Il est conçu pour une configuration légère et une mise en route rapide. (équivalent : webpack)
- **Vue 3 Composition API** est une nouvelle façon d'organiser la logique des composants en utilisant des fonctions. Elle permet de découper la logique en "*composables*" réutilisables. Les composables sont des fonctions réutilisables qui regroupent la logique associée à un aspect spécifique du composant, tels que les données, les méthodes, les effets, et bien plus encore.
- **Pinia** est une bibliothèque de gestion d'état basé sur la Composition API qui gère l'état global de l'application en utilisant des "*stores*" qui contiennent des données partagées et des méthodes pour les manipuler.
- **API fetch** est utilisée pour effectuer des requêtes HTTP dans l'application. Elle est native dans les navigateurs modernes et permet d'interagir avec des ressources distantes (comme des API REST) de manière asynchrone. (alternative à Axios)
- **Vue-router** est une bibliothèque de routage. Elle permet de gérer la navigation dans l'application en fonction de l'URL de la page. Vous pouvez définir des routes et associer des composants à chaque route. Lorsque l'URL change, le composant associé est rendu à l'emplacement approprié de votre application.

Architecture



Retour d'expérience

J'ai suivi les cours officiel de la VueSchool. Ils sont très bien fait. Je n'ai pas d'éléments de comparaison avec d'autres frameworks front mais j'ai trouvé Vue très facile, pratique et bien pensé. La documentation est très clair. Vue 3 et les nouveaux outils offrent une réelle amélioration positive qui simplifie largement le développement pour avoir pu les comparer avec Vue 2.

- **ce qui fonctionne :**
  - l'affichage d'appel get de l'API event-hub-server
- **ce qui a été difficile :**
  - comprendre le principe de réactivité
- **ce qui serait intéressant d'ajouter :**
  - des appels post, des formulaires…
  - l'authentification, les routes protégés et les permissions utilisateurs
  - les templates html css, pour lesquels j'ai peu de connaissances.
  - des effets, l'utilisation de bibliothèques vueUse et vuetify
  - Le framework Quasar fonctionne sur VueJS et permet un déploiement multi-plateforme avec des templates html et css. Il semble donc idéal avec un apprentissage réduit si on connait VueJS. Nuxt à l'air très bien aussi..

Sources utiles

- Vue School : https://vueschool.io/
- Documentation officielle : https://vuejs.org/
- Pinia : https://pinia.vuejs.org/
- Vue Router : https://router.vuejs.org/
- Code cheatsheet : https://learnvue.co/LearnVue-Vue-3-Cheatsheet.pdf
- Quasar : https://quasar.dev/
- Nuxt : https://nuxt.com/

Architecture finale du projet frontEnd BackEnd

Architecture finale

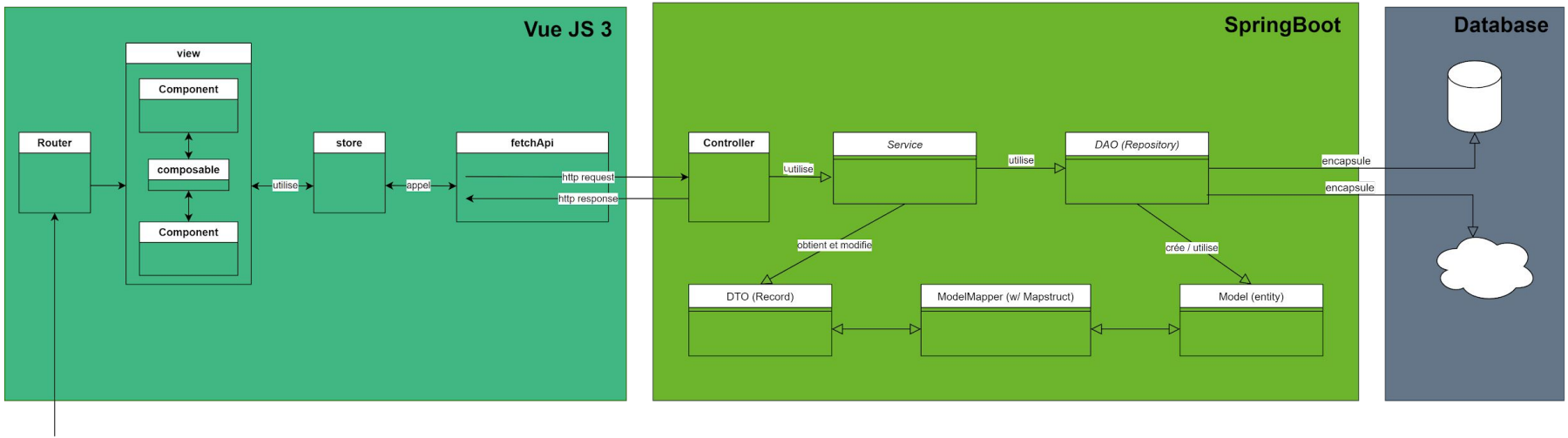
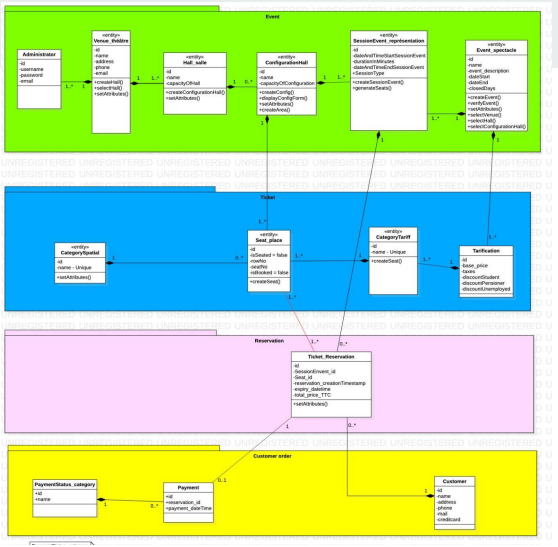


Schéma de la base de données



https://github.com/vifeng/TicketToTheMoon/blob/8b79a39f2db0c1ac14d44a67fb4448c8f70daf88/documentation/Database/TTTM%20db%20V8.jpg

Démonstrations

version longue focus sur le backend:

version courte focus sur le frontend : https://youtu.be/hJsLGcKklNU

Les commandes de ces démonstrations sont disponibles sur le wiki :

https://github.com/vifeng/TicketToTheMoon/wiki

Merci

Merci pour votre attention
Merci aux professeurs du Cnam qui m'ont accompagnés sur ce projet : M. Graffion, M. Rosmorduc
Merci aux professeurs du Cnam pour leurs enseignements et leur bienveillance.

Site de billetterie en ligne appelé "Event Hub"

Code : https://github.com/vifeng/TicketToTheMoon

Virginie Fengarol